amcs

PARALLEL DYNAMIC PROGRAMMING ALGORITHMS:
MULTITRANSPUTER SYSTEMS

JAN SADECKI*

* Department of Electrical Engineering and Automatic Control, Technical University of Opole
ul. Sosnkowskiego 31, 45–233 Opole, Poland
e-mail: jsad@po.opole.pl

The present paper discusses real parallel computations. On the basis of a selected group of dynamic programming algorithms, a number of factors affecting the efficiency of parallel computations such as, e.g., the way of distributing tasks, the interconnection structure between particular elements of the parallel system or the way of organizing of interprocessor communication are analyzed. Computations were implemented in the parallel multitransputer SUPER NODE 1000 system using from 5 to 50 transputers.

Keywords: dynamic programming, parallel computations, transputers, multitransputer systems, parallel optimization algorithms

1. Introduction

The present paper discusses problems associated with investigations the possibilities, ways and efficiency of the application of dynamic programming methods to solving dynamic optimization problems in parallel multitransputer systems. Optimization computations were implemented on the multitransputer SUPER NODE 1000 system. The obtained results served to illustrate many essential problems associated with implementation of parallel computations. In general, they refer to the influence of the parallel system structure and the manner of organizing the interprocessor communication upon the global efficiency (speedup) of parallel computations. In a particular way, dynamic programming methods are suitable for conducting such investigations, since they offer real possibilities of parallelizing computations on different levels of algorithms and, consequently, in the distribution of tasks, obtaining parallelism of a different granularity and various communication requirements. The presented results were obtained during investigations carried out by the author at the Centre for Mathematical Software Research of the University of Liverpool.

2. Parallel Systems and Computations

When planning parallel computations, it is possible to distinguish three basic groups of problems associated with the architecture of parallel systems, formulation of parallel computational algorithms and implementation of computations in a concrete parallel system. In the classification formulated by Flynn (1972), multitransputer systems are in principle contained in the systems of the MIMD type (Multiple Instruction stream—Multiple Data stream). In general, digital systems, which belong to this group, are built from two processors or a larger number of processor units with comparable properties in which, depending on the way the system is organized, all processors have access to common memory (common memory systems), or they are furnished with local private memories (distributed memory systems). At the same time, these system processor units perform functions of node elements in the generated networks responsible for communication with other elements, and for directly interconnected data exchange between the elements for which they constitute indirect nodes. In that case, an important problem is the way of connecting (topology) particular elements in more complex structures. The following types of structures belong to the typical, most often applied solutions in this range: linear chain, ring, square, tree or hypercube (Kozielski and Szczerbiński, 1993; Sadecki, 2001).

MIMD systems with distributed memory have become relatively popular recently. This results from the possibility of connecting a large or very large, coming up to several thousand, number of cheap processor elements in parallel complex structures. This technique, called "massive parallelism", allows again and again to achieve a good computing speedup in spite of increasing, together with an increment in the number of connected processors, the system load due to the use of communication and synchronization mechanisms based on message transfer.

In the 1980s and 1990s multitransputer systems (Harp, 1989; Wysocki and Kwolek, 1994; TAN, 1989) (eg. the Super Node 1000 system (Interi, 1991)) became very popular in Europe. A transputer is an integrated circuit, made in the VLSI technology, designed to be utilized in parallel data processing. The name itself, resulting from combining TRANSPUTER=TRANSmitter+comPUTER, emphasizes that this device is a one-circuit computer in which utmost attention is paid to both computational and communication problems. Special hardware solutions found in it are used in the form of a communication system handling 4 bidirectional, serial links utilized for direct connections with other transputers. These connections serve the construction of multiprocessor systems, implementing real parallel computations. The interfaces of 4 transputer links operate independently of other links and of the processor. In that case the processor, after initiating a communication task, can proceed to performing tasks associated with implementation of the successive process.

Parallel processing, and in fact its substitute in the form of concurrent processing implemented by the distribution of processor time, can be performed in a single transputer. This is possible owing to the universal programming language OCCAM, worked out for the needs of parallel processing, taking message transfer into consideration. Many functions which in other processors must be emulated by means of software or executed by other external devices are implemented by hardware in a transputer. Hence transputers can also be applied successfully to processing in real time. In a large family of transputers which have been made by the British company INMOS since 1985 there are a number of hardware solutions which, having been improved, are characterized by better and better possibilities in the range of both computing and communication speeds. IMS T212, IMS T414, IMS T800 and IMS T9000 are the basic members of this family. In Fig. 1, as an example, a general scheme of the inner architecture of the transputer IMS T800 is presented.

This transputer contains: a 32-bit processor with 64-bit coprocessor capacity at the clock frequency of 30 MHz amounting to 15 MIPS and 3.3 MFLOPS, 4 KB internal memory, 4 bidirectional serial interfaces with the transmission speed of 10/20 Mbit/s and external memory to 4GB. Four bidirectional serial interfaces, with which each transputer is equipped, are utilized for creating complex multitransputer structures via direct connections of many different transputers. This task can be made much easier by the application of special link-switching circuits designed for this purpose, such as, e.g., switch chip C004 or switch chip C104, designed for cooperation with transputers T9000 (TAN, 1989; Wysocki and Kwolek, 1994). Transputer T9000 is characterized by much better param-
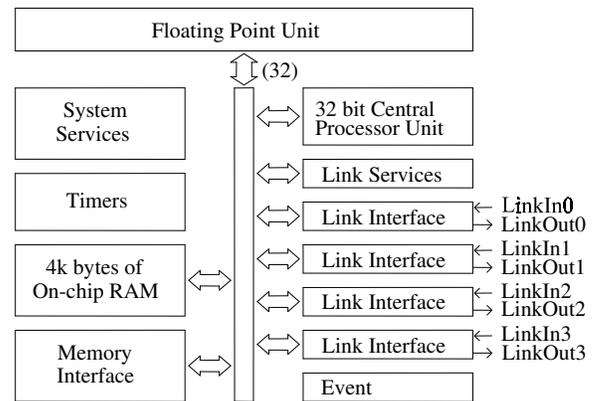
Fig. 1. General scheme of the architecture of transputer T800.

eters than T800. It is composed of closely connected together units: a 32-bit arithmetic-logic unit (ALU) and a 64-bit floating-point unit (FPU) of computing power with the clock frequency of 50 MHz of the order of 200 MIPS and 25 Mflops (peak power) and 70 MIPS and 15 Mflops (long-lasting power). It is equipped with a 16 KB internal memory, 4 bidirectional serial interfaces with the transmission rate to up 100 Mbit/s, a virtual channel processor and external memory to up 4 GB.

A natural programming language of transputers is OCCAM, a procedural concurrent programming language worked out by INMOS (Occam 2, 1998; Wysocki and Kwolek, 1994). It is characterized by exceptional simplicity, allowing concise and effective applications for both individual transputers and multitransputer systems to be created. OCCAM is a language enabling a program to be presented as a set of processes which operate concurrently and intercommunicate through program channels. On the other hand, in the case of a multitransputer system, this communication consists in message transfer between processes implemented by various transputers. The complex implementation of a number of tasks associated with compilation, configuration and putting programs into operation, written in the OCCAM language, can be performed with the use of various kinds of tools. The following operating systems are often applied here: HELIOS, TDS (Transputer Development System), OCCAM-TOOLSET or ANSI C-TOOLSET (TDS, 1988).

In the SUPER NODE system and in many other multitransputer systems, owing to the application of special switch systems, it is practically possible to implement any structure of connections. This task, implemented with the use of a program, consists in a mutual assignment of four particular links of transputers entering into the composition of a generated structure. The possibility of configuring the system is particularly important when an optimized choice of the system structure is analyzed for the solved problems and applied algorithms.

The SUPER NODE 1000 system works under the Unix-like Operating System IDRIS. It can contain from 16 to 1024 transputers, working at the maximum speed of up to 1.600 Mflops. This system, in the version installed at the Centre for Mathematical Software Research (University of Liverpool), is composed of 65 transputers of the 4 MB T800 type, 24 transputers of the 256 KB T800 type and 16 transputers handling the VCR (Virtual Channel Router). The virtual channel system permits (from the viewpoint of software) direct implementation of transmissions between any pair of transputers occurring in a configurated multitransputer structure (Debbage *et al.*, 1991).

As for various advantages resulting from the parallel way of data processing, we can mention the possibility of a considerable computing speedup and the qualitatively new possibilities of controlling rapid dynamic processes, as well as simulation of real complex processes in technology (Sadecki, 1996; 1999). The concept of parallel computations is utilized for the description of a situation in which many processor units, controlled centrally, work at the same time to solve one problem. Before implementing parallel computations, it is required to formulate a suitable algorithm ensuring that the intentional expectations are fulfilled concerning, e.g., the computing speed, and that the possibility of the applied hardware is employed in an optimal way. The adaptation of a given algorithm to implementation in a parallel system consists generally in identifying a set of independent subtasks in it, which can be solved in parallel, securing intercommunication when computations are made. A set of such subtasks is called a parallel algorithm. In general, the efficiency of implementation of parallel algorithms in a real system will depend on the following factors:

- the architecture of a parallel system in which it is implemented,
- the way of interchanging information between the processor units,
- a suitable distribution of data in the elements of system memory,
- a proper distribution of tasks between processor units.

In the distribution of tasks, the best situation is created by the possibility of equally loading all processors in time. However, it is not always possible to satisfy this condition. Generally, one can distinguish two basic ways of distributing:

- static distribution of tasks: tasks are allocated once before computations,
- dynamic distribution of tasks: tasks are allocated to processors on-line, while the algorithm is implemented.

In practice, algorithms for dynamically balancing the load on particular processors, implemented in a dispersed manner by all the processor elements of the system, are also employed. These are relaxation algorithms, which successively implement tasks tending towards equalizing the load on all processors (Baker and Milner, 1991). They constantly monitor each of the processors, their state and their load (i.e., the number of tasks which are still to be performed), and the state of the load on processors directly connected with them in the system.

The analysis associated with formulating and comparing parallel algorithms requires the use of a uniform measure selected according to the properties of these algorithms. Most often it is carried out on the basis of the so-called computing speedup factor (Brochard, 1989)

$$S(N, P) = \frac{T(N, 1)}{T(N, P)}, \tag{1}$$

where P denotes the number of processor units utilized in computations, whereas N characterizes the numerical quantity of the problem, i.e. its dimension or the number of the processed data elements.

One should theoretically understand a time $T(N, 1)$ as the time of the best existing sequential algorithm or the time necessary for performing the algorithm adopted as a model one. In practice, it often means the time of implementation on one processor of a sequential algorithm, submitted then to parallelisation or, if possible, the time of implementation on one processor of a parallel algorithm. On the other hand, $T(N, P)$ denotes the implementation time of the analyzed parallel algorithm with the use of P processors.

3. Parallel Dynamic Programming Algorithms

Computations concerning the problems analyzed in this paper were implemented in the parallel multitransputer SUPER NODE 1000 (SNODE) system. They were performed with the use of OCCAM and they are associated with the efficiency analysis of different parallel implementations of dynamic programming algorithms, as well as the analysis of the influence of the parallel system structure and the way of organizing interprocessor communication upon the global efficiency (speedup) of computing. The presented deliberations were based on some examples of dynamic optimization problems which can be formulated in the following way: A control process is given and it is described by the system of state equations:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}_0\big[\boldsymbol{x}(t), \boldsymbol{u}(t), t\big], \ \ t_0 \leq t \leq t_K, \ \ \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \ \ (2)$$

where $\boldsymbol{x}$ is the n-dimensional state vector ($\boldsymbol{x} \in \mathbb{R}^n$), $\boldsymbol{u}$ denotes the m-dimensional control vector ($\boldsymbol{u} \in \mathbb{R}^m$),

and $\boldsymbol{f}$ stands for the n-dimensional vector function (non-linear in general). The performance criterion is defined in the form of the functional

$$J(\boldsymbol{x}, \boldsymbol{u}) = \int_{t_0}^{t_K} l_0\big[\boldsymbol{x}(t), \boldsymbol{u}(t), t\big]\,\mathrm{d}t + \Psi\big[\boldsymbol{x}(t_K), t_K\big], \quad (3)$$

where l_0 is a scalar cost function and Ψ signifies a scalar terminal cost function.

Moreover, some restrictions are imposed on the state and control variables which can be generally formulated as the following relations:

$$\begin{aligned} \boldsymbol{x}(t) &\in \Omega_x(t), & \Omega_x &\subset \mathbb{R}^n, \\ \boldsymbol{u}(t) &\in \Omega_u\big[\boldsymbol{x}(t), t\big], & \Omega_u &\subset \mathbb{R}^m. \end{aligned} \quad (4)$$

The optimization task consists in finding a control vector $\boldsymbol{u}(t)$ such that if (2) and (4) are satisfied, it minimizes the performance criterion (3).

Application of a discrete version of the dynamic programming method (DP) for solving the above problem requires its prior discretization. By dividing the interval $\langle t_0, t_K \rangle$ into K subintervals of equal length of $\Delta t = (t_K - t_0)/K$ $(t = t_0 + k\Delta t, \quad k = 0, 1, \ldots, K)$, the problem (2)–(4) can be reformulated in a discrete form. On the other hand, the state equations, a performance criterion and constraints will be respectively determined by

$$\begin{aligned} \boldsymbol{x}(k+1) &= \boldsymbol{f}\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big], \\ k &= 0, 1, \ldots, K-1, \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \end{aligned} \quad (5)$$

$$J\big[\boldsymbol{x}(k), \boldsymbol{u}(k)\big] = \sum_{k=0}^{K-1} l\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big] + \Psi\big[\boldsymbol{x}(K), K\big], \quad (6)$$

$$\begin{aligned} \boldsymbol{x}(k) &\in \Omega_x[k], & \Omega_x &\subset \mathbb{R}^n, \quad k = 0, 1, \ldots, K, \\ \boldsymbol{u}(k) &\in \Omega_u\big[\boldsymbol{x}(k), k\big], & \Omega_u &\subset \mathbb{R}^m, \quad k = 0, 1, \ldots, K-1, \end{aligned} \quad (7)$$

where

$$\boldsymbol{f}\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big] = \boldsymbol{x}(k) + \boldsymbol{f}_0\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big]\Delta t,$$

$$l\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big] = l_0\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big]\Delta t.$$

In this case, the optimization task consists in searching for a control sequence $\{\boldsymbol{u}(0), \boldsymbol{u}(1), \ldots, \boldsymbol{u}(K-1)\}$ satisfying (5) and (7) and minimizing the value of the performance criterion (6). Application of the DP method to solving the problem (5)–(7) is based on making use of the principle of optimality (Findeisen *et al.*, 1980). This principle was formulated by Bellman (1957) for a wide range of systems whose future behaviour can be fully (or statistically) determined on the basis of the knowledge of their present state. For the problems formulated above, it can be expressed as follows (with the assumption that the optimal control exists):

The optimal strategy has a property such that regardless of what the initial state or initial control would be, the remaining controls must form the optimal strategy from the viewpoint of the state resulting from the first fragment of the control trajectory.

A discrete version of the DP method can be applied to solve either discrete by nature or discretized continuous optimization problems. The application of the principle of optimality to solving the problem described by the relations (5)–(7) leads to a recursive procedure for determining optimal control, which, in a mathematical notation, assumes the form of the iterative functional equation

$$\begin{aligned} I\big[\boldsymbol{x}(k), k\big] = \min_{\boldsymbol{u}(k) \in \Omega_u} \Big\{ &l\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big] \\ &+ I\big[\boldsymbol{f}\big[\boldsymbol{x}(k), \boldsymbol{u}(k), k\big], k+1\big] \Big\}, \quad (8) \end{aligned}$$

where $k = 0, 1, \ldots, K-1$, and $I[\boldsymbol{x}(K), K] = \Psi[\boldsymbol{x}(K), K], \boldsymbol{x}(k) \in \Omega_x$. Here $I[\boldsymbol{x}(k), k]$ denotes the so-called minimum cost function, defined as follows:

$$\begin{aligned} I\big[\boldsymbol{x}(k), k\big] = &\min_{\boldsymbol{u}(j) \in \Omega_u, j=k, k+1, \ldots, K-1} \\ &\left\{ \sum_{j=k}^{K-1} l\big[\boldsymbol{x}(j), \boldsymbol{u}(j), j\big] + \Psi\big[\boldsymbol{x}(K), K\big] \right\}. \quad (9) \end{aligned}$$

This defines the minimum cost which can be obtained when admissible controls are considered for the final segment of the trajectory starting at an arbitrary point $\boldsymbol{x}(k) \in \Omega_x, \quad k = 0, 1, \ldots, K-1$.

One of the advantages of the discrete DP method is the possibility of including a wide class of constraints imposed on both state and control variables. These constraints do not complicate the computations and can lead to a decrease in the computational requirements associated with implementation of the method. They determine the areas of admissible states and admissible controls denoted by $\Omega_x(k)$ and $\Omega_u(\boldsymbol{x}, k)$, respectively. In numerical implementation of the DP method, the process of solving (8) requires prior discretization of the state variables $\boldsymbol{x}(k)$ and, with an enumerative approach to the minimization process, discretization of the control variables $\boldsymbol{u}[\boldsymbol{x}(k), k]$ as well (Larson, 1968; Sadecki, 1987).

The process of solving the functional equation (8) consists in determining the values of the function $I[\boldsymbol{x}(k), k]$ and those of the optimal control $\hat{\boldsymbol{u}}[\boldsymbol{x}(k), k]$

at all the discrete points of the state space $\Omega_x(k)$, with k varying from $k = K - 1$ to $k = 0$. Then, on the basis of the computed values of $\hat{\boldsymbol{u}}[\boldsymbol{x}(k), k]$ for given $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and for k varying from $k = 0$ to $k = K$, it is possible to assign the optimal trajectory $\hat{\boldsymbol{x}}(k)$ and the corresponding optimal control $\hat{\boldsymbol{u}}(k)$, $k = 0, 1, \ldots, K - 1$, which is a solution to the problem (5)–(7). Since the computational requirements associated with solving (8) are, particularly for multidimensional problems, incomparably greater than those associated with computing the optimal trajectory from (5), they decide about the time-consumption of the dynamic programming method. Hence, when parallel implementation of this method is considered, most attention is paid to solving (8) (Sadecki, 1987).

Considerable computational requirements of the DP method, increasing with the dimension of the optimization problem and significant requirements of this method concerning the computer memory, have stimulated the author to seek new, amended versions of the DP algorithm. Apart from the basic version of the dynamic programming method discussed above, there is a wide group of its variants characterized by much better properties with respect to the requirements of the computational process (Larson, 1968). Some of these algorithms do not differ practically in their applications from the basic version of the DP method. The majority of them, however, are formulated for a limited, strictly determined class of optimization problems. One of such methods, permitting the requirements to decrease considerably with respect to storage, is the state increment dynamic programming method (SIDP) (Larson, 1968; Sadecki, 1987). In this method the value of increment δt along the time variable t is not constant as in the basic method (where this increment is Δt), but can vary, assuming currently the values determined by the dependence:

$$\delta t = \min \left\{ \min_{i=1,2,\ldots,n} \left[\frac{\Delta x_i}{|f_i(\boldsymbol{x}, \boldsymbol{u}, t)|} \right], \quad \Delta t \right\}, \quad (10)$$

where Δx_i is the increment resulting from discretizing the i-th component of $\boldsymbol{x}$, Δt is the increment resulting from discretizing time t, and δt means the time interval for which the control assumes a constant value. It is determined as the minimum time in which one of the state variables forming the vector $\boldsymbol{x}$ changes about its full increment Δx_i resulting from discretizing the area Ω_x.

For the SIDP method, the functional equation (8) assumes the following form:

$$I(\boldsymbol{x}, t) = \min_{\boldsymbol{u} \in \Omega_u} \left\{ l(\boldsymbol{x}, \boldsymbol{u}, t)\delta t \right.$$

$$\left. + I\big[\boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)\delta t, t + \delta t\big] \right\}. \quad (11)$$

Assigning the values of δt on the basis of the relation (10) assures that in the section of δt, the quantity

x_i, $i = 1, 2, \ldots, n$ can be altered at most as much as Δx_i. At the same time, the relation $\delta t \leq \Delta t$ holds. Consequently, the value of $I[\boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)\delta t, t + \delta t]$ can be determined only on the basis of the value of the minimum cost function defined in the immediate vicinity of the point $\boldsymbol{x}$ in which current computations are made, i.e. in the area $x_i - \Delta x_i \leq x_i \leq x_i + \Delta x_i$, $i = 1, 2, \ldots, n$ (for the stage of $t + \Delta t$ and/or the stage of $t + 2\Delta t$). The presented approach permits the area of $\Omega_x(t)$ to be divided into $(n + 1)$-dimensional subdomains, called blocks, of the minimum width with respect to the variable x_i, which is $2\Delta x_i$. In practice, the recommended width with regard to t covers the range from $5\Delta t$ to $15\Delta t$. In the computations made within one block, it is necessary to store the value of the function $I(\boldsymbol{x}, t)$, determined in the same block, for one or several (according to the applied interpolation and extrapolation procedures) "previous" time stages, which considerably restricts storage requirements in comparison with the basic DP algorithm. A detailed discussion of the method of computations, made in blocks and on their boundaries, is presented in (Larson, 1968; Sadecki, 1987; 1992). In Fig. 2 a simplified diagram of computations implemented for the conventional DP method and for the SIDP algorithm is presented.

(a) conventional DP algorithm   (b) SIDP algorithm
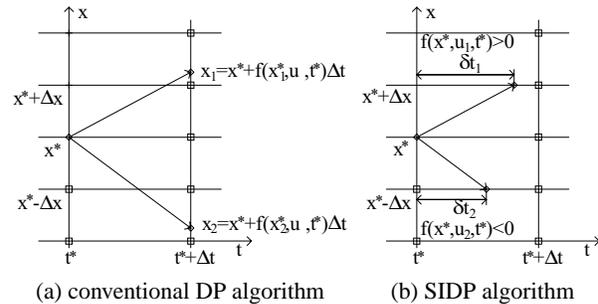
Fig. 2. Diagram of computations implemented at a given point (x^*, t^*) for the conventional DP method and for the SIDP method (at $n = 1$, $m = 1$).

The idea of parallel data processing creates a basis for further considerable relaxing of the requirements of the DP method with respect to the computation time, through their distribution to many processor units and with regard to memory requirements, via distribution of data between local memory modules. When analyzing the procedure used for solving the functional equation (8), one can notice that it consists of computations performed in three basic iteration loops:

(i) in relation to the index of stage $k = K - 1, K - 2, \ldots, 2, 1, 0$,

(ii) with regard to all the discrete values of the state vector $\boldsymbol{x}(k) \in \Omega_x(k)$, determined at a given stage k,

(iii) in respect to all the discrete values of the control vector (an enumerative approach) $\boldsymbol{u}[\boldsymbol{x}(k), k] \in \Omega_u[\boldsymbol{x}(k), k]$, determined at a given discrete point $\boldsymbol{x}(k) \in \Omega_x(k)$.

The above observation is a basis for the formulation of different parallel versions of the dynamic programming method, among which one can distinguish two basic ones (Sadecki, 1987):

1. Parallel state algorithm (PSA): parallelizing computations within loop (ii),

2. Parallel control algorithm (PCA): parallelizing computations within loop (iii).

As the least portion of tasks which can be distinguished in these algorithms we consider a set of operations associated with the computations of the values on the right-hand side of (8) at one discrete point of the state space—the PSA algorithm, and a set of operations connected with the computations of the values on the right-hand side of this equation at one discrete point of the state space and for one discrete value of the control vector—the PCA algorithm. These tasks are allocated to particular processors at the beginning of the computing process with the use of the static distribution method.

By formulating and analyzing the efficiency of parallel implementations of the DP method, use is made of three basic parameters determining the volume of a discrete optimization task, as well as a parameter determining the number of processors utilized in computations. The following symbols are adopted:

N – the number of discrete values of the state vector $\boldsymbol{x}$, specified in the set $\Omega_{x(k)}$, with the assumption that N does not depend on k,

M – the number of discrete values of the control vector $\boldsymbol{u}$, specified in the set $\Omega_{u[\boldsymbol{x}(k),k]}$, with the assumption that M depends on neither $\boldsymbol{x}$, nor k,

K – the number of time stages ($k = 0, 1, \ldots, K - 1$),

P – the number of processors employed in the computations.

The PSA and PCA algorithms can be formulated as follows (Casti *et al*, 1973; Malinowski and Sadecki, 1986; 1990; Sadecki, 1987; Sadecki and Galewicz, 1991):

Parallel state algorithm (PSA):

(i) Each processor calculates the values of $I[\boldsymbol{x}(k), k]$ for N/P discrete values of the vector $\boldsymbol{x}(k)$.

(ii) Each processor sends the computed values of $I[\boldsymbol{x}(k), k]$ to all the other processors.

(iii) Steps (i) and (ii) are repeated for all time stages.

Parallel control algorithm (PCA):

(i) Each processor calculates the values of $I[\boldsymbol{x}(k), k]$ for M/P discrete levels of the vector $\boldsymbol{u}[\boldsymbol{x}(k), k]$ (all the processors compute at the same point $\boldsymbol{x}(k)$), choosing a locally optimal value of $\hat{\boldsymbol{u}}^*[\boldsymbol{x}(k), k]$ and the corresponding value of $\hat{I}^*[\boldsymbol{x}(k), k]$.

(ii) Each processor sends the computed values of the function $\hat{I}^*[\boldsymbol{x}(k), k]$ and those of control $\hat{\boldsymbol{u}}^*[\boldsymbol{x}(k), k]$ to the other processors in order to choose globally optimal values of $\hat{\boldsymbol{u}}[\boldsymbol{x}(k), k]$ and $\hat{I}[\boldsymbol{x}(k), k]$.

(iii) Steps (i) and (ii) are repeated for all discrete values of $\boldsymbol{x}(k)$ and all time stages.

If in the above algorithms $(N \bmod P) = 0$ or, respectively, $(M \bmod P) = 0$ occurs, then all the processors taking part in computations will be loaded with tasks uniformly. Otherwise, some processors will be loaded with one task more than the others.

Taking the PSA as an example, it is possible to formulate parallel implementation for the SIDP method. To this end, we propose to divide the area of Ω_x into blocks, the number of which will be equal to the number of the processors utilized in computations. Thus each processor will implement computations in one block of the time width $K\Delta t$. Some explanation is necessary here regarding computations made on the boundaries of the blocks. One of the possible approaches is the method consisting in including, when the computations are made in a given block (on its boundary), only such discrete controls $\boldsymbol{u}$ for which the value of $\boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)\delta t$ lies within the same block. The other controls $\boldsymbol{u}$ will be included in computations on the same boundary but from the side of an adjacent block (Sadecki, 1987). However, in a parallel implementation, such a solution would require a wider use (as compared with the sequential version) of extrapolation procedures. In the parallel implementation presented in this paper a somewhat different approach is used. Practically, it does not differ from a sequential implementation of the algorithm. It introduces some delay, usually one time stage, in implementation of computations within particular blocks. Thus, for example, processor P_1, after performing computations for the given stage, sends the values of the function I computed at the boundary points to processors allocated to the neighbouring blocks. This is necessary to begin computations in these blocks and, at the same time, it is a signal to commence computations by these processors for the same k, when processor P_1 already initiates computations for the successive stage $k + 1$. Thus the values of I for the boundary points are computed only by one of the processors associated with the adjacent blocks. However, this algorithm still requires implementation of more communication tasks associated with sending some values of I, calculated by particular processors in the immediate vicinity of the block boundaries for the "earlier" stage $k + 1$, necessary for a correct implementation of the SIDP method. Some delay introduced within this algorithm makes the efficiency of the parallel algorithm dependent upon the number of stages K, whereas its influence upon the computational speedup will decrease together with the increment of K.

In this case, the parallel SIDP algorithm can be formulated in the following way:

Parallel state increment dynamic programming algorithm:

(i) All processors, in succession, begin the computing process of the values of $I[x(k), k]$ for N/P discrete levels of vector $\boldsymbol{x}(k)$ occurring within blocks assigned to them with time shift, concerning blocks in its immediate vicinity, equal to 1.

(ii) Each processor sends the computed values of $I[\boldsymbol{x}(k), k]$ for $\boldsymbol{x}(k)$, lying on the block boundary, to the processors associated with blocks in its immediate vicinity.

(iii) Steps (i) and (ii) are repeated for all time stages.

All of the above algorithms were implemented in the parallel SNODE system. At the same time, use was made of the possibilities, inherent to the system, of practically configuring in any way. This forms a basis for the analysis associated with examination of the influence of the system configuration upon the efficiency of computations concerning parallel implementation of dynamic programming algorithms. As has already been mentioned, the SNODE system is equipped with the so-called virtual channel router (Debbage *et al*, 1991), permitting direct communication to be organized between any pair of system elements (point-to-point), even if they are not physically connected with each other, without any necessity of programming communication between intermediate system nodes, and thus somehow independently of its real configuration. This functionality facilitates the programming of the communication tasks between any system elements, since in reality, in order to implement a concrete transmission task, it is sufficient to determine the number of a destination processor (or a source one) independently of the place in which this processor is situated, and to define data which are subject to transmission. The application of a virtual channel router resulted in slowing down the whole system and in some slowing down of data transmission in comparison with the real capabilities of the transputers. Communication time between processors not connected directly is significantly longer than the transmission time between the processors having such connections. As a consequence, the efficiency of the algorithms taking full advantage of the point-to-point communication is different than that of the algorithms using communications tasks concerning data transmission only between directly connected processors. In such a case, the influence upon the efficiency of parallel implementations of the analyzed algorithms will be exerted not only by the system structure, but also by the way of organizing the communication tasks.
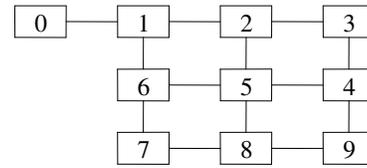
Communication processes for processor P_i ($i = 1, 2, \ldots, P$)

send: $\quad P_i \xrightarrow{\boldsymbol{I}_i} P_j, \quad j = 1, 2, \ldots, i-1, i+1, \ldots, P,$

receive: $\quad P_i \xleftarrow{\boldsymbol{I}_j} P_j, \quad j = 1, 2, \ldots, i-1, i+1, \ldots, P.$

Fig. 3. Square structure ($P = 9$) with a market scheme of "full-exchange", implemented between each pair of system elements (operations *send* and *receive* mean sending and receiving data, respectively).

For analysis, three basic types of system configuration are considered: square (Fig. 3), linear chain (Fig. 4) and ring (Fig. 5). In these figures we also show the way of implementing the data exchange concerning transmission of some fragments of a vector $\boldsymbol{I}$ formed from the values of the function $I[\boldsymbol{x}(k), k]$ computed by particular processors between those processors. In the square structure, the total number of processors is $P = P_x P_y$, where P_x and P_y denote the numbers of processors in a row and in a column of the square considered, respectively. When referring to this type of structure, it is assumed that $P_x = P_y$, since for this variant the lowest value of the maximal number of interprocessor connections between the most distant processors in the structure (which is generally equal to $(P_x - 1) + (P_y - 1)$) is achieved. In each of the above-mentioned structures, only processors with numbers $1, 2, \ldots, P$ were used in the implementation of the studied algorithms. The processor with number 0 is almost exclusively utilized to manage the distribution of tasks and resources, synchronization of time, and possible collection of the results of computations. On the other hand, communication tasks were implemented with the use of one of the following three algorithms: communication of the "full-exchange" type, direct communication implemented on the basis of a linear chain structure, when only direct connections between processors were used, and communication of the "master-slave" type.

Communication of the "full-exchange" type

This algorithm consists in implementing bidirectional data transmission between each pair of elements, aiming to create, in each of these elements, a full copy of all the results computed parallelly by particular processors. This manner of communication is utilized by the PSA. Denoting by $\boldsymbol{I}_i$ the vector of the values of $I[\boldsymbol{x}(k), k]$, determined while solving (8) by processor P_i, it is assumed that this processor sends the whole vector $\boldsymbol{I}_i$, locally calculated by itself, to all the other processors, obtaining in turn the values $\boldsymbol{I}_j$, $j = 1, 2, \ldots, i-1, i+1, \ldots, P$, as

computed by the remaining processors. From the software viewpoint, utilizing possibilities which are offered by the application of virtual channels to direct communication between any pair of system elements, such communication tasks can be schematically formulated for the i-th processor as follows:

$$
\begin{aligned}
&\text{PAR}\\
&\quad \text{PAR } j = 1, 2, \ldots, i-1, i+1, \ldots, P\\
&\qquad \text{send}\,(P_j, \boldsymbol{I}_i)\\
&\quad \text{PAR } j = 1, 2, \ldots, i-1, i+1, \ldots, P\\
&\qquad \text{rec}\,(P_j, \boldsymbol{I}_j).
\end{aligned} \tag{12}
$$

In order to obtain a clear notation, a simplified diagram of the PAR construction with a replicator is applied here. It is taken from the OCCAM language but it diverges a little from the general convention of this language. The construction PAR $j = 1, 2, \ldots, P$ in (12) conventionally means that all the processes specified in successive rows with a two-space indentation, identified by the value of index j, will be implemented in parallel. On the other hand, $\text{send}\,(\cdot, \cdot)$ and $\text{rec}\,(\cdot, \cdot)$ are communication procedures implementing tasks of sending and receiving data, respectively, at the same time; the first of the parameters of these procedures determines the numbers of destination and source processors whereas the second parameter identifies data subjected to transmission.

As a result of the communication task (12) carried-out by all processors P_i, $i = 1, 2, \ldots, P$, each of them will have its own copy of the whole vector $\boldsymbol{I} = [\boldsymbol{I}_1, \boldsymbol{I}_2, \ldots, \boldsymbol{I}_P]^T$. In practice, however, it may appear that in order to make the required computations at a given discrete point $\boldsymbol{x}(k) \in \Omega_x$ at stage k it is not necessary to remember all the values of this function for stage $k + 1$. In such a case, only those values of the function $I[\boldsymbol{x}(k+1), k+1]$ that are really indispensable to computations can be transmitted between processors and only for those processors which need these values. Such a constrained manner of exchange, called optimal communication, was applied within the PSA. How many values of the function I and to which processors they should be sent depend on several factors such as, e.g., the form of the state equations, the assumed discretization of the state variable $\boldsymbol{x}(\Delta x_i, i = 1, 2, \ldots, n)$, and the assumed value of the step Δt, as well as the number of the processors used in computations. In general, in order to perform computations at any point $\boldsymbol{x}^*(k)$ of the set $\Omega_x(k)$, it is essential to remember the values of the function $I[\boldsymbol{x}(k+1), k+1]$ only for such discrete points $\boldsymbol{x}(k+1)$ belonging to the set $\Omega_x^*(k+1) = \{\boldsymbol{x}(k+1) : \boldsymbol{x}(k+1) = \boldsymbol{f}[\boldsymbol{x}^*(k), \boldsymbol{u}(k), k], \boldsymbol{x}^*(k) \in \Omega_x(k), \boldsymbol{x}(k+1) \in \Omega_x(k+1), \boldsymbol{u}[\boldsymbol{x}^*(k), k] \in \Omega_u\}$. If for each $\boldsymbol{x}^*(k) \in \Omega_x(k)$ the equality $\Omega_x^*(k+1) = \Omega_x(k+1)$ is satisfied, then communication will assume the form of "full exchange",

whereas if points such as $\boldsymbol{x}^*(k) \in \Omega_x(k)$ are numerous and other conditions, namely $\Omega_x^*(k+1) \subset \Omega_x(k+1)$ and $\Omega_x^*(k+1) \neq \Omega_x(k+1)$ are for them satisfied, then—according to the remarks made above—communication can be implemented in a more effective form. As will be shown, optimal communication is very effective, but it requires to assign *a priori* which of the values of $\boldsymbol{I}_i$, computed by particular processors P_i, should be sent to which of the other processors and from which processors, and what values should be received. Since the processors communicating with each other cannot in general have direct connection, the actual way of communication was implemented with the use of virtual channels.

Communication of the "master-slave" type

In communication implemented according to the "master-slave" scheme, each processor (slave) P_i, $i = 1, 2, \ldots, P$ transmits all the values of the function I ($\boldsymbol{I}_i$), calculated locally by itself, to the coordination processor (master) whose part can be played, e.g., by the processor denoted by number 0 in Fig. 3 (or every other processor in the system). The coordinating processor, after receiving all the values of $\boldsymbol{I}_i$, $i = 1, 2, \ldots, P$, sends either the full vector of the values of the function I or its fragment required by particular processors to the remaining processors. Communication tasks defined in such a way (making use of the virtual channel) can be schematically formulated for processor P_0 and processors P_i, $i = 1, 2, \ldots, P$ in the following way:

$$
\begin{aligned}
&\text{for processor } P_0:\\
&\quad \text{SEQ}\\
&\qquad \text{PAR } i = 1, 2, \ldots, P\\
&\qquad\quad \text{rec}\,(P_i, \boldsymbol{I}_i)\\
&\qquad \text{PAR } i = 1, 2, \ldots, P\\
&\qquad\quad \text{send}\,\big(P_i, (\boldsymbol{I}_1, \boldsymbol{I}_2, \ldots, \boldsymbol{I}_{i-1}, \boldsymbol{I}_{i+1}, \ldots, \boldsymbol{I}_P)\big),
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
&\text{for processors } P_i, \ i = 1, 2, \ldots, P:\\
&\quad \text{SEQ}\\
&\qquad \text{send}\,(P_0, \boldsymbol{I}_i)\\
&\qquad \text{rec}\,\big(P_0, (\boldsymbol{I}_1, \boldsymbol{I}_2, \ldots, \boldsymbol{I}_{i-1}, \boldsymbol{I}_{i+1}, \ldots, \boldsymbol{I}_P)\big).
\end{aligned}
$$

The construction SEQ, taken from the OCCAM language, means that all the processes specified in successive rows with a two-space indentation will be carried out sequentially (in succession).

Direct communication (via line)

In the case of direct communication, data transmissions are implemented with the use of only direct interprocessor connections. This algorithm can be practically implemented in each of the structures specified above. However, within the linear chain structure it must occur in two

cycles, differing in the direction of the information flow (Fig. 4). At the same time, the cycles can be arranged in a sequential order with respect to each other, or they can be implemented, wholly or at least partially, in parallel (according to the structure of interprocessor connections, e.g., as a single or a double chain). The maximum number of direct connections between the most distant processors in the chain is $P - 1$. Direct communication is applied with respect to the process of the parallel implementation of the PSA and the PCA.
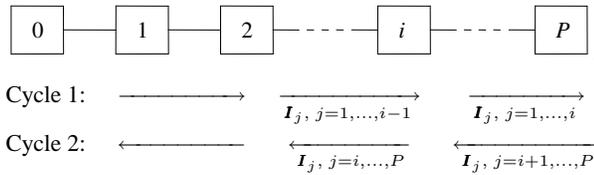
Fig. 4. Linear chain structure with a marked diagram of direct exchange for the PSA.

Parallel state algorithm (PSA):

Cycle 1: Each processor P_i $(i = 1, 2, \ldots, P - 1)$ sends the values of the minimum cost function $(\boldsymbol{I}_i)$, calculated by itself, and the values of this function obtained from processor $P_{i-1}(\boldsymbol{I}_j,\ j = 1, 2, \ldots, i - 1)$ to processor P_{i+1},

Cycle 2: Each processor P_i $(i = 2, 3, \ldots, P)$ sends the values of the minimum cost function $(\boldsymbol{I}_i)$, computed by itself, and the values of this function obtained from processor P_{i+1} $(\boldsymbol{I}_j,\ j = i + 1, i + 2, \ldots, P)$ to processor P_{i-1}.

The communication task defined in such a way can be schematically formulated for the i-th processor as follows (Fig. 4):

PAR
  SEQ
    rec $\left(P_{i-1}, (\boldsymbol{I}_j,\ j = 1, 2, \ldots, i - 1)\right)$
    send $\left(P_{i+1}, (\boldsymbol{I}_j,\ j = 1, 2, \ldots, i - 1, i)\right)$ (14)
  SEQ
    rec $\left(P_{i+1},\ (\boldsymbol{I}_j,\ j = i + 1, i + 2, \ldots, P)\right)$
    send $\left(P_{i-1},\ (\boldsymbol{I}_j,\ j = i, i + 1, i + 2, \ldots, P)\right)$.

In order to limit the number of cycles of data exchange occurring in the PCA, its implementation is modified. Namely, it is assumed that the data exchange will be implemented not every time after ending computations at successive discrete points of the set $\Omega_x(k)$, but collectively after making computations at all N discrete points of this area at a given stage k. In such an implementation the number of cycles of data exchange will be K (as in the PSA) and not NK, as in the basic implementation of this algorithm. An exchange algorithm in the

PCA will work in two cycles. The first cycle will implement a search for the best solution from among those determined by particular processors, whereas the task of the second one will be to send the optimal values of the function $I[\boldsymbol{x}(k), k]$ to all processors.

Parallel control algorithm (PCA):

Cycle 1: Each P_i $(i = 1, 2, \ldots, P - 1)$ receives from P_{i+1} N locally optimal values of controls of $\hat{\boldsymbol{u}}_{i+1}^*[\boldsymbol{x}(k), k]$ and N corresponding values of the function $\hat{I}_{i+1}^*[\boldsymbol{x}(k), k]$, determined at all discrete points of the set $\Omega_x(k)$ at a given stage k. Next, the values are compared with own results $(\hat{\boldsymbol{u}}_i^*[\boldsymbol{x}(k), k],\ \hat{I}_i^*[\boldsymbol{x}(k), k])$, better results are chosen with respect to the adopted criterion and then they are sent to P_{i-1}. Consequently, P_1 will hold the globally optimal solution ($\hat{\boldsymbol{u}}[\boldsymbol{x}(k), k]$, $\hat{I}[\boldsymbol{x}(k), k]$),

Cycle 2: Each P_i $(i = 2, 3, \ldots, P)$ receives N optimal values of the minimum cost function $\hat{I}[\boldsymbol{x}(k), k]$ from P_{i-1}, and then sends them to P_{i+1} (the values are indispensable to begin computations at the successive stage).

The communication tasks presented above, determined for the PCA, can be schematically formulated for the i-th processor as follows:

SEQ
  rec $\left(P_{i+1}, \left(\hat{I}_{i+1}^*[\boldsymbol{x}(k), k], \hat{\boldsymbol{u}}_{i+1}^*[\boldsymbol{x}(k), k]\right)\right)$,
  min $\left[(\hat{I}_{i+1}^*, \hat{\boldsymbol{u}}_{i+1}^*), (\hat{I}_i^*, \hat{\boldsymbol{u}}_i^*)\right] \rightarrow (\hat{I}_i^*, \hat{\boldsymbol{u}}_i^*)$
  send $\left(P_{i-1}, \left(\hat{I}_i^*[\boldsymbol{x}(k), k], \hat{\boldsymbol{u}}_i^*[\boldsymbol{x}(k), k]\right)\right)$ (15)
  rec $(P_{i-1}, \hat{\boldsymbol{I}})$
  send $(P_{i+1}, \hat{\boldsymbol{I}})$.

Direct communication is also applicable to the PSA (Fig. 5) and to the PCA, implemented in the system with a ring structure. In this structure communication is implemented in one cycle. For example, for the PSA, each P_i $(i = 1, 2, \ldots, P)$ sends to P_{i-1} at first the values of $\boldsymbol{I}_i$ computed by itself and next the values of $\boldsymbol{I}_j,\ j = i + 1, i + 2, \ldots, P, 1, 2, \ldots, i - 1$ received in succession from processor P_{i+1}. Communication ends when each of the processors has collected all the values of the function I. The maximum direct connections between
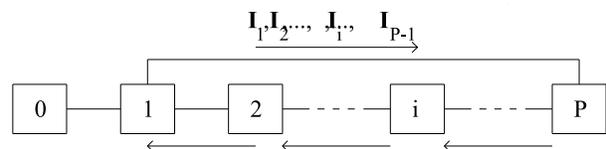
Fig. 5. Ring structure with a marked diagram of exchange.

the most distant processors in a P-element ring are $0.5P$, rounded down when the number of processors is odd.

In the communication algorithms discussed above, implemented in the parallel system, it is possible to execute generally more than one communication task at the same time. For example, for the PSA method implemented in a linear chain structure it is possible to execute two transmission tasks (for $P \geq 3$) at the same time (data transmission in two different directions). For the ring structure and the same method, the number of such tasks is P in general. Moreover, it is possible to parallelize some communication operations in relation to computational tasks. If, for example, the processors differ in the execution time of computational tasks (which often happens in practice), then a processor receiving data from another processor which has earlier executed its computational task can implement the task of data reception parallelly with the executed computational task. For example, a portion of the algorithm (14), supplemented by "computations" ensuring such a possibility, can be presented with some simplification as follows:

SEQ
  PAR
    computations   (16)
    rec $\big(P_{i-1},\ (\boldsymbol{I}_j,\ j = 1, 2, \ldots, i-1)\big)$
   send $\big(P_{i+1},\ (\boldsymbol{I}_j,\ j = 1, 2, \ldots, i-1, i)\big)$.

Below, numerical results obtained by practical implementation of the presented methods and parallel algorithms are presented. The estimation of the efficiency of parallel computations is carried out on the basis of a speedup factor determined by the relation (1). Denoting the implementation time of the DP algorithm on one transputer by t_{SEQ} and, at the same time, parallel implementation time of this algorithm by t_{PAR} with the use of P transputers, the value of the speedup factor is determined as

$$S(P) = \frac{t_{\mathrm{SEQ}}}{t_{\mathrm{PAR}}}. \qquad (17)$$

The computations presented in this paper were performed using the example which follows.

4. Computational Example

The dynamic optimization problem concerns the system

$$\dot{x}(t) = u(t), \quad 0 \leq t \leq 10, \qquad (18)$$

where

$$0 \leq x(t) \leq 8, \quad -2 \leq u(t) \leq 2, \quad x(0) = 8. \qquad (19)$$

We should find control and state trajectories so as to minimize the value of the performance criterion

$$\min : J = \int_0^{10} \big[x^2(t) + u^2(t)\big]\,\mathrm{d}t + 2.5\big[x(10) - 2\big]^2. \qquad (20)$$

The application of a discrete version of the DP method to solve the above problem requires its discretization in time, i.e. presenting it in the form of the equations (5)–(7):

$$x(k+1) = x(k) + u(k)\Delta t,$$
$$k = 0, 1, \ldots, K-1, \quad K = 10/\Delta t + 1, \qquad (21)$$

$$0 \leq x(k) \leq 8, \quad k = 0, 1, \ldots, K, \quad x(0) = 8,$$
$$-2 \leq u(k) \leq 2, \quad k = 0, 1, \ldots, K-1, \qquad (22)$$

$$\min : J = \sum_0^{K-1} \big[x^2(k) + u^2(k)\big]\Delta t + 2.5\big[x(K) - 2\big]^2. \qquad (23)$$

Furthermore, assuming the increments of Δx and Δu, one should also discretize the variables $x(k)$ and $u(k)$, obtaining N and M discrete levels, respectively, for each of these variables, where

$$N = (8/\Delta x) + 1, \quad M = (4/\Delta u) + 1. \qquad (24)$$

In order to solve such a discretized problem, we can directly apply the parallel algorithms.
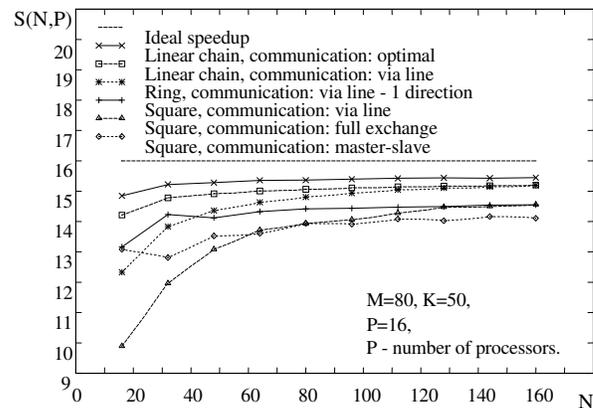
Fig. 6. Parallel state algorithm (PSA).

In Fig. 6 the results obtained by means of parallel implementation of the PSA with $P = 16$ transputers are presented. This figure shows the values of a computing speedup factor as the function of N, where N denotes the number of discrete values of the state variable x (for constant values of $M = 80$ and $K = 50$),

since for the PSA the value of N determines the number of tasks allocated to particular processors. The plots are given for all the configurations and ways of communication discussed above. As can be seen, the best results are achieved for the linear chain structure and data exchange, determined above by the notion of optimal communication. The results obtained for the master-slave communication are also presented. However, this manner of computations appeared comparatively the least effective for transputer systems. In general, one should emphasize—which is very promising at the same time—that the values of the computing speedup obtained for the best variants of the parallel algorithms achieve a high level for 16 processors, reaching the value of 15.5.
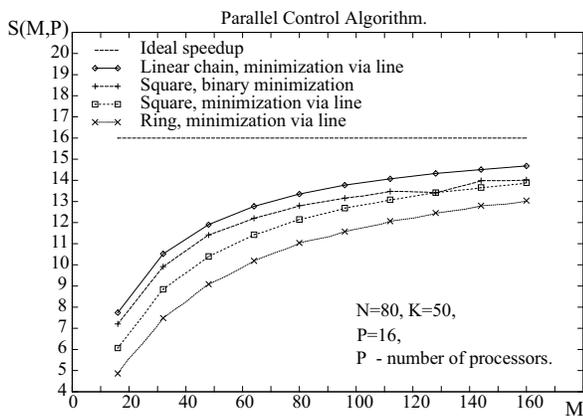
Fig. 7. Parallel control algorithm (PCA).

In Fig. 7 the results obtained by means of a parallel implementation of the PCA with the same number of $P = 16$ transputers are presented. This figure shows the values of the computing speedup factor as a function of M, where M denotes the number of discrete values of the control variable u (for constant values of $N = 80$ and $K = 50$), since for the PCA the value of M determines the number of tasks allocated to particular processors. These diagrams concern the presented configurations and ways of communication, whereas in the PCA the search for the optimal values of $\hat{u}$ and $\hat{I}$ is implemented together with the exchange of data. The best results are obtained here, similarly to the case of the PSA, for the linear chain structure. The diagrams provided with the comment "minimization via line" correspond to the discussed implementation of the PCA method for direct communication. On the other hand, some comment is required by the diagram denoted by "binary minimization". In this method, the choice of the best solution from among those obtained by particular processors is implemented by executing transmission tasks and comparing the local solutions between the determined pairs of processors. At the same time, many such tasks can be executed at a given mo-

ment. It is assumed that each of the processors, after performing computation tasks allocated to it at a given stage of the algorithm, will send the local values of $\hat{u}^*$ and $\hat{I}^*$ as found by itself (each time computing and choosing a better solution by the processor receiving data), in turn, to the first, second, fourth, eighth, sixteenth, etc. processor in the chain (i.e. processor P_i will send data to processors P_{i+1}, P_{i+2}, P_{i+4}, P_{i+8}, etc.). A complete implementation of such a manner of communication requires a programmed closure of the chain structure into a ring one. As a result of exchange, after $l = \log_2 P$ cycles (rounded up if P is not an integer power of 2) each of the processors will have the optimal values of $\hat{u}$ and $\hat{I}$. This algorithm requires ensuring the possibility of communication between any pair of the system elements, which was achieved by using the virtual channel router.

The values of the computing speedup obtained for such a manner of communication are not much worse than those achieved for the linear chain structure with direct communication. In general, the results obtained for the PCA are clearly worse than those achieved with the application of the PSA (the values of the computing speedup for the PCA method achieved the level of 14.6 with 16 processors, which is not a bad result). This generally results from the fact that in the PCA method, together with data exchange, the algorithm of choosing the best solution is implemented.
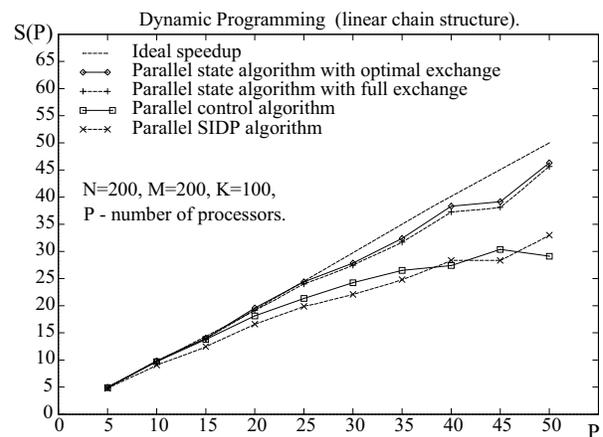
Fig. 8. Comparison of parallel DP algorithms.

In Fig. 8, for some versions of the algorithms discussed above, the relations of the computing speedup factor as a function of the number of employed transputers for constant values of $N = 200$, $M = 200$ and $K = 100$ are presented. At the same time, a much higher number of transputers than before is used, namely, $P = 5, 10, 15, 20, \ldots, 50$. In much the same way as before, the PSA turned out to be comparatively better (the computing speedup with $P = 50$ processors achieved the value of $S = 47$). The parallel SIDP algorithm approxi-

mates the PCA in respect of the achieved speedup. Some non-uniformity in diagrams ("steps") results from the fact that it is not possible to obtain a uniform distribution of tasks for every number of processors used in computations.

As can be seen from the presented results for both the PSA and PCA, the best outcomes in terms of the achievable real computing speedup are obtained when the algorithms are implemented in the system of a linear chain structure, with the information interchange constrained to data transmission only between the system elements which have a direct connection. Hence structures of this type and communication are adopted as fundamental ones in the further analysis associated with more detailed computations concerning the investigation of the efficiency of the parallel implementation of the DP method.

The investigations were concerned with the PSA, the PCA, the parallel SIDP algorithm and the PSA with optimized communication with respect to the amount of transmitted data. The results are presented in Figs. 9–14, with Figs. 9–11 concerning the PSA and PCA and Figs. 12–14 referring to the SIDP algorithm. The PSA with optimal communication is denoted by PSA-O in the figures. These figures represent the influence of the number of processors used for computations upon the computing speedup for different values of parameters N, M and K determined by the quantities of the adopted digitizing steps for the state variable Δx, the control variable Δu and the stage variable (time one) Δt.

The alterations to the quantities N and M are of essential importance for both the PSA and PCA. In the case of the PSA, the change in M is associated with alteration to the number of computations (time consumption) performed by particular processors when preserving unchanged, in this case, communication requirements associated with the value of N in this method. On the

Fig. 10. The PSA and PCA algorithms: $S = S(P)$, $N = 100$, $K = 100$, $M = 100, 200$.
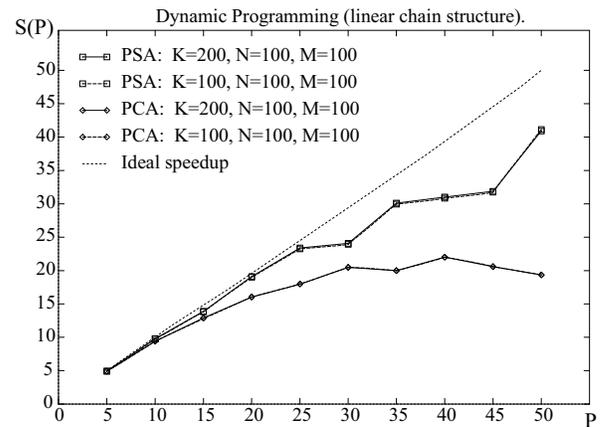
Fig. 11. The PSA and PCA algorithms: $S = S(P)$, $N = 100$, $M = 100$, $K = 100, 200$.
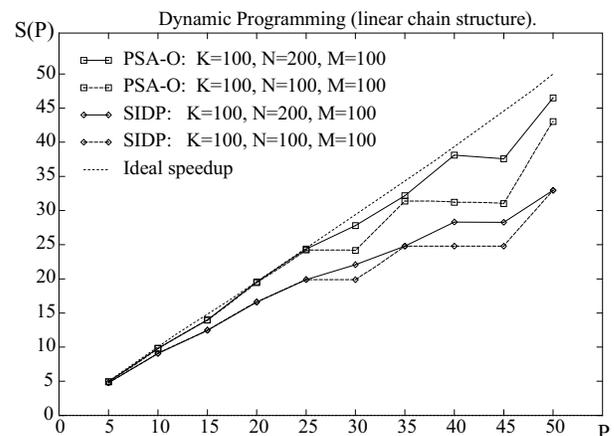
Fig. 12. The parallel SIDP algorithm and the PSA with optimal communication (PSA-O): $S = S(P)$, $M = 100$, $K = 100$, $N = 100, 200$.

Fig. 9. The PSA and PCA algorithms: $S = S(P)$, $M = 100$, $K = 100$, $N = 100, 200$.
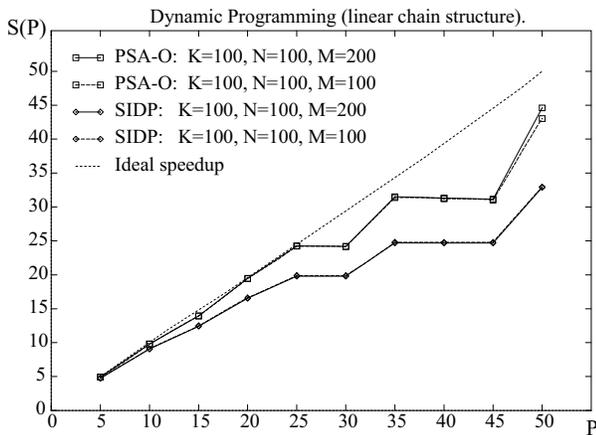
Fig. 13. The parallel SIDP algorithm and the PSA with optimal communication (PSA-O): $S = S(P)$, $N = 100$, $K = 100$, $M = 100, 200$.
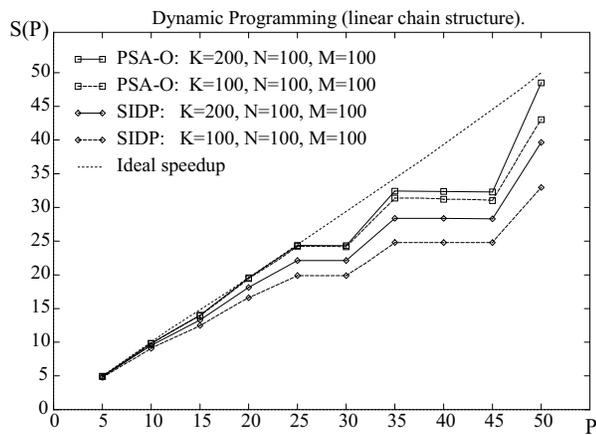
Fig. 14. The parallel SIDP algorithm and the PSA with optimal communication (PSA-O): $S = S(P)$, $N = 100$, $M = 100$, $K = 100, 200$.

other hand, the change in N brings about a change in the number of computations implemented by particular processors, together with a change (having the same direction) in communication requirements. In the case of the PCA, together with data exchange, the choice of the best solution is implemented. This data exchange can be implemented in two ways depending on its place in the algorithm, namely, each time after executing the computations at given discrete point in the set $\Omega_x(k)$, or collectively after computing at all the discrete points of the set Ω_x at a given stage k. In principle, these variants are identical with respect to the volume of exchanged data. They differ, however, in the number of exchange cycles occurring at every stage. If in the first variant, at every stage, one should perform N exchange cycles, whereas in the second variant only one cycle, then the second method is

applied to computations as being more effective. The essential difference between the PSA and the PCA consists in determining the task distribution. In the case of the PSA method, this distribution takes place on the highest level of the algorithm (a larger granulation of the local tasks), whereas in the case of the PCA method—on a lower level (a smaller granulation of the local tasks).

As a supplement to the analyzed results, some additional explanations can be provided. In order to concentrate on the configuration and communication problems, the exemplary computations presented in this paper concern only a one-dimensional optimization problem in both state and control variables. However, in general, this does not restrict the analysis since, as results from both the theoretical reasons and previous investigations, the efficiency of parallel DP algorithms generally depends on the values of N, M (and P), while the dimensions of vectors $\boldsymbol{x}$ and $\boldsymbol{u}$ for which those values were determined are not so much significant (Sadecki, 1987). As can be seen from the presented diagrams, the influence of the values of K, N, M or P upon the computational speedup factor is very significant and, at the same time, it depends on the version of the adopted parallel algorithm. For example, the effect of the value of K on the computing speedup of the parallel PSA and PCA algorithms is negligible in practice, but it is very significant for the parallel SIDP algorithm and the PSA algorithm with optimal communication (Sadecki, 1987). On the other hand, the effect of the value of M on the computing speedup of the parallel SIDP and the PSA with optimal communication is negligible, but it is very significant for the parallel PSA and PCA algorithms. In general, the best results were obtained for the most complicated algorithm as regards the organization of interprocessor communication, namely the PSA with optimal communication. Next there are the STN and SIDP algorithms. The worst results were obtained for the PCA; this generally results from the fact that for that algorithm, the data exchange is implemented together with the choice of the best solution.

The above single-dimensional results have been confirmed in the analysis made for more complex examples, involving multidimensional systems in both state and control variables (Sadecki, 2002).

A natural supplement to the discussed results are Figs. 15–17, in which the values of time for parallely solving the task from the example on the multitransputer SNODE system considered are presented for the PSA and PCA methods (with different values of K, N, M and P). These results, taken in conjunction with those presented in Figs. 9–14, give a whole picture of computational requirements for the parallel algorithms under consideration.

Parallel State Algorithm (linear chain structure).

K=var, N=100, M=100
K=100, N=var, M=200
K=100, N=200, M=var
P=50
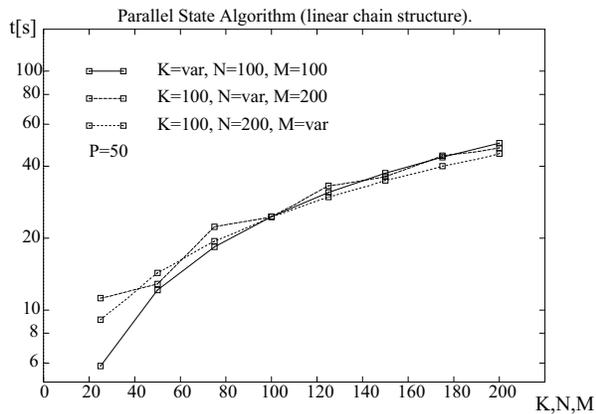
Fig. 15. Computation time $t[s]$ for the PSA: $t = t(N)$, $t = t(M)$, $t = t(K)$, $P = 50$.

Parallel Control Algorithm (linear chain structure).

P=50
K=var, N=100, M=100
K=100, N=var, M=200
K=100, N=200, M=var
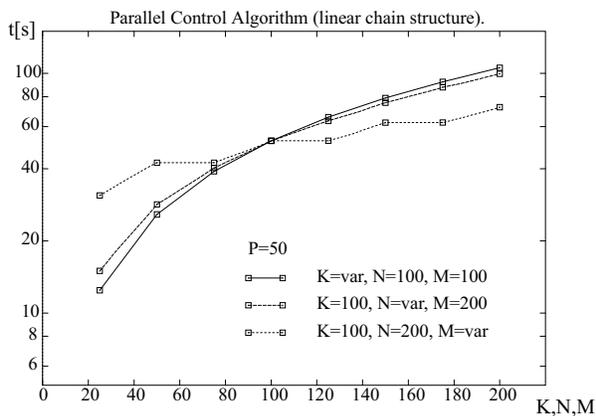
Fig. 16. Computation time $t[s]$ for the PCA: $t = t(N)$, $t = t(M)$, $t = t(K)$, $P = 50$.

Dynamic Programming (linear chain structure).

PCA: K=100, N=200, M=200
PSA: K=100, N=200, M=200
PCA: K=100, N=100, M=100
PSA: K=100, N=100, M=100
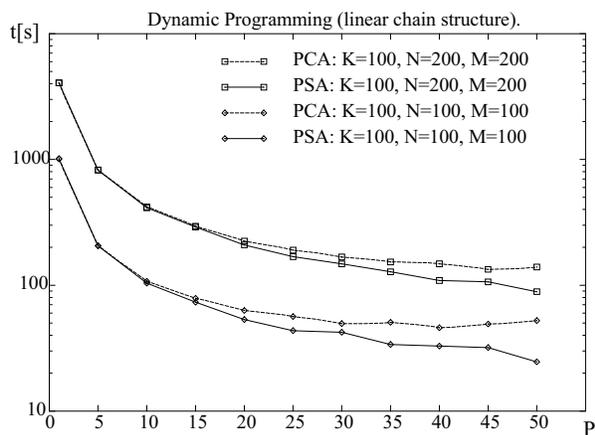
Fig. 17. Computation time $t[s]$ for the PSA and the PCA: $t = t(P)$, $N = 100, 200$, $M = 100, 200$, $K = 100$.

5. Conclusions

This paper is concerned with the investigation of the possibilities of real implementations of a selected group of

parallel dynamic programming algorithms. Some examples of parallel implementations of these algorithms in the multitransputer SNODE 1000 system of different configurations of connections between its elements as well as many different problems arising in computations of this type are presented. It is shown that both the proper choice of the system structure and the way of organizing interprocessor communication can considerably affect the efficiency of parallel computations and consequently the computing speedup factor. In principle, the obtained results point out very serious possibilities of improving the efficiency of implementation of the DP method through its parallelization. This also demonstrates the suitability of this method for parallelizing. In many cases, the obtained values of the computing speedup factor are not considerably different from the value of $S(P) = P$, theoretically the best one to be obtained. The discussed models of parallel computations can be easily applied to other optimization methods and algorithms, and to other types of numerical algorithms, since in parallel implementations many of these algorithms consist of cycles of computations and data exchange. At the same time, the exchange often consists of communication tasks, aiming at creating complete copies of certain sets of data allocated in parallel by the other processors present in the system.

References

Baker S.A. and Milner K.R. (1991): *Performance monitoring and dynamic load balancing, ESPRIT Project 2701.* — Royal Signals and Radar Establishment, Malveren, UK.

Bellman R. (1957): *Dynamic Programming.* — Princeton: Princeton Univ. Press.

Brochard L. (1989): *Efficiency of some parallel numerical algorithms on distributed systems.* — Parallel Comput., Vol. 12, No.1, pp. 21–44.

Casti J., Richardson M. and Larson R. (1973): *Dynamic programming and parallel computers.* — JOTA, Vol. 12, No. 4, pp. 423–438.

Debbage M., Hill M. and Nicole D. (1991): *Virtual channel router, Ver. 2.0, User guide, ESPRIT Project 2701.* — University of Southampton.

Findeisen W., Szymanowski J. and Wierzbicki A. (1980): *Theory and Computation Methods of Optimization.* — Warsaw: Polish Scientific Publishers, (in Polish).

Flynn M.J. (1972): *Some computer organizations and their effectiveness.* — IEEE Trans. Comp., Vol. C–21, No. 9, pp. 948–960.

Harp G. (1989): *Transputer Applications.* — London: Pitman Publishing.

Interi G. (1991): *Using the SN1000.* — Liverpool: Liverpool University Press.

Kozielski S. and Szczerbiński Z. (1993): *Parallel Computers: Architecture, Elements of Programming*. — Warsaw: WNT, (in Polish).

Larson R. (1968): *State Increment Dynamic Programming*. — New York: Elsevier.

Malinowski K. and Sadecki J. (1986): *Dynamic programming: A parallel implementation*, In: Parallel Processing Techniques for Simulation (Singh M.G., Allidina A.Y. and Daniels B.K., Eds). — New York: Plenum Press, pp. 161–170.

Malinowski K. and Sadecki J. (1990): *Parallel implementation of dynamic programming methods in multiprocessor systems of different structures: Analysis of efficiency*. — Archives of Automatic Control and Remote Control Engineering, Vol. XXXV, No. 3–4, pp. 119–140.

Occam 2 (1988): *Occam 2, Reference Manual*. — London: INMOS Ltd.

Sadecki J. (1987): *Parallel implementation of dynamic programming methods in multiprocessor systems and investigation of their efficiency*. — Ph.D. Th., Warsaw University of Technology (in Polish).

Sadecki J. and Galewicz St. (1991): *Parallel computations in real two-processor system: Dynamic programming method*. — Archives of Automatic Control Engineering and Robotics, Vol. XXXVI, No. 1, pp. 193–203.

Sadecki J. (1992): *Possibilities of speedup of optimization computations by their implementation in parallel two-processor system: Decomposition algorithms in dynamic programming*. — Scientific Papers of the Higher School of Eng. in Opole, Poland, Electrical Engineering, No. 35, pp. 5–27 (in Polish).

Sadecki J. (1996): *Parallel optimization algorithms of complex systems and hierarchical control: Parallel distributed memory systems*. — Research project carried out for the State Committee for Scientific Research in Poland, No. 3 P403 02706, Final Report, Higher School of Eng. in Opole, Poland, pp. 142 (in Polish).

Sadecki J. (1999): *The analysis of efficiency of parallel implementation of some optimization algorithms*. — Proc. 13-th Nat. Conf. *Automatic Control*, Opole, Poland, pp. 341–344 (in Polish).

Sadecki J. (2001): *Parallel Optimization Algorithms and Investigation of Their Efficiency: Parallel Distributed Memory Systems*. — Studies and Monographs, Technical University of Opole, Opole, Poland (in Polish).

Sadecki J. (2002): *The analysis of efficiency of parallel implementation of selected two-level optimization algorithms*. — Multitransputer Syst. (to appear).

TAN (1989): *The Transputer Applications Notebook, System and Performance*. — Melksham, Wiltshire: Redwood Press Ltd., INMOS Ltd.

TDS (1988): *Transputer Development System*. — London: Prentice Hall, INMOS Ltd.

Wysocki M. and Kwolek B. (1994): *Parallel Computations and Transputers in Automatic Control*. — Rzeszów: Technical University Press (in Polish).