PERIODICA POLYTECHNICA SER. CHEM. ENG. VOL. 43, NO. 1, PP. 3–16 (1999)

PARALLELIZATION OF ITERATIVE DYNAMIC PROGRAMMING (IDP)

Frank HARTIG, Klaus MANDEL and Frerich J. KEIL

Technical University of Hamburg-Harburg
Eissendorfer Strasse 38
D–21071 Hamburg, Germany
EMAIL: keil@tu-harburg.de

Received: April 6, 1999

Abstract

The present paper describes an algorithm of the parallelization of Iterative Dynamic Programming by the ‘Parallel Virtual Machine’ (PVM) language. It will be demonstrated that the parallelization can lead to a considerable reduction in computing time.

Keywords: Iterative Dynamic Programming (IDP), parallelization, Parallel Virtual Machine (PVM), optimization.

1. Introduction

Parallel processing is a fast growing technology that influences many areas of engineering. Reviews about chemical engineering applications of parallel processing have been written by e.g. CLEMENTI et al. (1989), JAJA (1992), MCRAE (1990) and WILSON (1995). It comprises algorithms, computer architecture, programming and performance analysis. There is a strong interaction among these aspects.

Dynamic Programming (DP) has an inherent parallel structure that makes this algorithm especially suitable for parallelization. Therefore, some authors have suggested parallel algorithms of the conventional DP scheme (see BERTSEKAS and TSITSIKLIS (1989), SMITH (1993), CHEN (1986), EDMONDS et al. (1993), RYTTER (1988)). Parallelism has become necessary because single-processor computers are not powerful enough to solve the so-called Grand-Challenge problems of science and engineering, for example, the optimization of entire chemical plants. The availability of public domain programs like ‘Parallel Virtual Machine’ (PVM) (GEIST et al. (1993)) or ‘Message-Passing Interface’ (MPI) (GROPP et al. (1994)) enables every programmer to develop his own parallel programs, and any personal computer or work station may be connected with the aid of low-cost network interface cards. In this paper Iterative Dynamic Programming (IDP) will be parallelized by PVM.

2. Parallelization of Iterative Dynamic Programming

Iterative Dynamic Programming (IDP) will be used for the optimization of sequential processes with P stages as shown in *Fig. 1.*

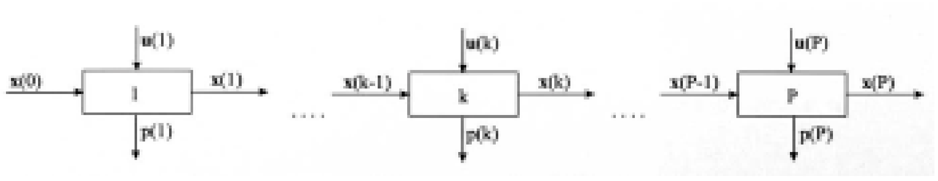


Fig. 1. State vectors $\mathbf{x}$, control vectors $\mathbf{u}$ and objective function p of each stage

In each stage k the state vector $\mathbf{x}(k)$ is a function $\mathbf{G}$ of the state vector $\mathbf{x}(k-1)$ and the control vector $\mathbf{u}(k)$

$$\mathbf{x}(k) = \mathbf{G}(\mathbf{x}(k-1), \mathbf{u}(k)). \tag{1}$$

The state vector $\mathbf{x}(0)$ at the entrance of the staged process is defined by the initial state vector $\mathbf{x}_0$:

$$\mathbf{x}(0) = \mathbf{x}_0. \tag{2}$$

The objective value $p(k)$ of each stage k is a function p_k of the state vector $\mathbf{x}(k-1)$ and the control vector $\mathbf{u}(k)$

$$p(k) = p_k(\mathbf{x}(k-1), \mathbf{u}(k)). \tag{3}$$

The objective value F of the complete staged process is equal to the sum of the objective values $p(k)$ and the value of the objective function g, which depends on the state vector $\mathbf{x}(P)$ at the exit of the staged process

$$F = \sum_{k=1}^{P} p_k(\mathbf{x}(k-1), \mathbf{u}(k)) + g(\mathbf{x}(P)). \tag{4}$$

The objective function is to be minimized by choosing optimal control vectors between the boundaries $\alpha(k) \leq \mathbf{u}(k) \leq \beta(k)$. Application of the dynamic programming computational procedure to high-dimensional nonlinear problems, defined by *Eqs.* (1) – (4), with continuous state and control vectors leads to the following difficulties:

1. Sufficiently fine grids are necessary for an accurate solution.
2. The computational procedure of dynamic programming calculates the following optimization problems

$$F_p(\mathbf{x}(P-1)) = \min_{\mathbf{u}(P)} \left[p_P(\mathbf{x}(P-1), \mathbf{u}(P)) + g(\mathbf{x}(P)) \right], \tag{5}$$

$$F_k(\mathbf{x}(k-1)) = \min_{\mathbf{u}(k)} \left[p_k(\mathbf{x}(k-1), \mathbf{u}(k)) + F_{k+1}(\mathbf{x}(k)) \right], \qquad k = P-1, ..., 1. \tag{6}$$

For each point $x^{[j]}(k)$ of the state grid k (see *Fig. 2*) the computational procedure stores the values of F_{k+1}, but for the optimization of the problems (6) the computational procedure needs also values of F_{k+1} for state vectors which are between the state grid points. These values can be calculated by an interpolation method, which has to be very accurate for not missing the global optimum.

3. If each state or control vector has r levels and the vector is n-dimensional, then a grid has r^n grid points. For high-dimensional problems n is high and therefore dynamic programming needs a very high number of grid points and a long calculation time as well as a large computer storage for the optimization of those problems.

LUUS (1989, 1990) has developed an algorithm named Iterative Dynamic Programming (IDP) which is based on dynamic programming and avoids these difficulties. The flow diagram of the basic IDP algorithm is given in *Fig. 2*. The grids of the control and state vectors are shown in *Fig. 3*. An example of the detailed FORTRAN program is given by LUUS (1989), and HARTIG and KEIL (1993a). The algorithm has been successfully applied to many global optimization problems of control theory and chemical reaction (LUUS and ROSEN (1991), LUUS and GALLI (1991), HARTIG and KEIL (1993b), LUUS (1993a, 1993b), HARTIG et al. (1995)).

In order to overcome the difficulty of sufficiently fine grids LUUS (1989) picked up an idea by BELLMAN and DREYFUS (1962) who suggested calculation of the problem again and again by reducing the size of the grids. First the problem will be calculated by using coarse grids. Then the mid points of the state and control grids are set equal to the preliminary optimal state and control vectors and the ranges of the grids are reduced. The second difficulty can be avoided by using a special procedure, developed by LUUS (1989) for calculating F_{k+1} for a state vector $\mathbf{x}(k)$ between the state grid points. IDP stores the optimal control vectors $\mathbf{u}_{opt}^{[j]}(k)$ for the state vectors $\mathbf{x}^{[j]}(k-1)$, $(k = P, ..., 1)$. The algorithm searches for the nearest state grid point $\mathbf{x}\,[j^*]\,(k)$ to $\mathbf{x}(k)$ and calculates *Eqs.* (1) and (3) of the next stage by using the optimal control vector stored for $\mathbf{x}^{[j^*]}(k)$. This procedure is repeated until the last stage P is calculated. The value of F_{k+1} is equal to the sum of the objective values $p(q)$, $q = k+1, ..., P$ and $g(\mathbf{x}(P))$.

BOJKOV and LUUS (1992) have shown that the third difficulty can be overcome by using a few randomly distributed control grid points. By using those randomly distributed control grid points, BOJKOV and LUUS (1992) calculated with IDP optimization problems with 100-dimensional state vectors.

The focus of this paper is the parallelization of IDP by using the parallel virtual machine (PVM) message-passing environment (MCBRIAN (1994), SUNDERAM et al. (1994), GEIST et al. (1993, 1994)).

```
Estimate the optimal control vectors $\mathbf{u}^*(k)$ of all stages ($k = 1, \ldots, P$) and define the search
Regions $\mathbf{u}^*(k) \pm \mathbf{r}(k)$. Define the number of state grid points $N$, number of control grid points
$R$, reduction factor $\gamma$ and the number of iterations.
REPEAT
    Generate control grids with $R$ grid points $\mathbf{u}^{[i]}(k)$ ($i = 1, \ldots, R$) for $k$ stages $k = 1, \ldots, P$.
    The control grid points are placed within the the regions $\mathbf{u}^*(k) \pm \mathbf{r}(k)$ and satisfying the
    boundary conditions $\alpha(k) \leq \mathbf{u}^{[j]}(k) \leq \beta(k)$.
    Generate state grids with $N$ grid points $\mathbf{x}^{[j]}(k)$, ($j = 1, \ldots, N$) for each stage $k = 1, \ldots, P-1$
    by calculating the staged process with the controls $\mathbf{u}^{[j]}(k)$, ($j = 1, \ldots, N$). The initial
    state vector $\mathbf{x}^{[1]}(0)$ is equal to $\mathbf{x}_0$.
    For $k = P$ to $1$ step $-1$
        For $j = 1$ to $N$ step $1$
            $F_{\min} = \infty$
            For $i = 1$ to $R$ step $1$
                $\mathbf{x} = \mathbf{G}(\mathbf{x}^{[j]}(k-1), \mathbf{u}^{[i]}(k))$
                $F = p_k(\mathbf{x}^{[j]}(k-1), \mathbf{u}^{[i]}(k))$
                For $q = k+1$ to $P$ step $1$
                    Search $j^*$ that fulfils $\{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(q-1) - \mathbf{x})^2\}$
                    $F = F + p_q(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1))$
                    $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1)$
                $F = F + g(\mathbf{x})$
                $F < F_{\min}$
                then
                    $\mathbf{u}_{\text{opt}}^{[j]}(k) = \mathbf{u}^{[i]}(k)$
                    $F_{\min} = F$
    $j^* = 1, F = 0, \mathbf{x} = \mathbf{x}_0$
    For $k = 1$ to $P$ step $1$
        $\mathbf{r}(k) = \gamma \mathbf{r}(k)$
        $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(k))$
        $F = F + p_k(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(k))$
        $\mathbf{u}^*(k) = \mathbf{u}_{\text{opt}}^{[j^*]}(k)$
        $k < P$
        then
            Search $j^*$ that fulfils $\{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(k) - \mathbf{x})^2\}$
    $F = F + g(\mathbf{x})$
UNTIL all iteration are executed
minimum value of the objective function: $F$ optimal controls: $\mathbf{u}^*(k)$, ($k = 1, \ldots, P$)
```

Fig. 2. Flow diagram of iterative dynamic programming

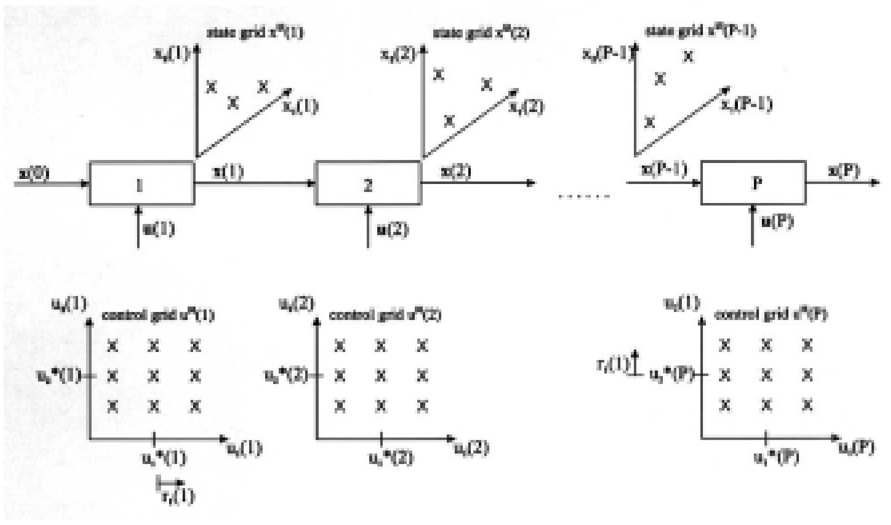


Fig. 3. State and control grids of iterative dynamic programming

The PVM software provides a unified framework within which parallel programs can be developed in an efficient manner using existing hardware. PVM enables a collection of heterogeneous computers to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures. The user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network, as well as communication and synchronization between tasks. At any point in the excecution of a concurrent application, any task in existence may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. The PVM system is composed of two main parts. The first part is a so-called daemon that resides on all the computers making up the virtual machine. The second part of the system is a library of PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message-passing, coordinating tasks, and modifying the virtual machine.

Because of its generality and its applicability to networks of workstations, PVM is one of the most widely used of all message-passing environments.

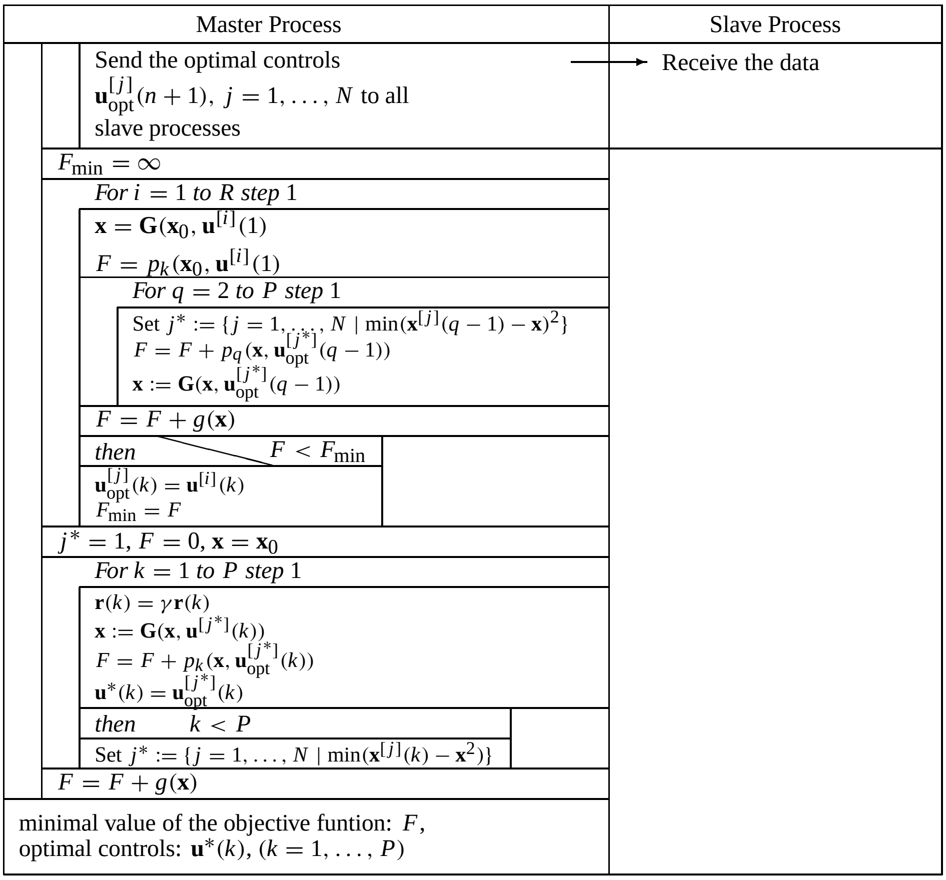
Flow diagram of the Parallel Iterative Dynamic Programming

Master Process	Slave Process
Load PVM subroutines and start the algorithm	
Start the slave processes	Load PVM subroutines and start the algorithm
Define search regions $\mathbf{u}^*(k) \pm \mathbf{r}(k)$, number of state grid points N and control grid points R, number of iterations, number of stages P and reduction factor γ	
Send the variables N, P and R to all slave processes $\rightarrow$	Receive the variables
do iteration $= 1$, number of iterations	
Send the variables $\mathbf{u}^*(k)$ and $\mathbf{r}(k)$ to all slave processes $\rightarrow$	Receive the variables and define the control grid points $\hat{\mathbf{u}}^{[j]}(k)$ $(j = 1, \ldots, N;\ k = 1, \ldots, P-1)$ for generating the state grids
do $j = 1$, number of slave processes	
Start a non-calculating slave process for the calculation of the state grid points $\mathbf{x}^{[j]}(k)$, $(k = 1, \ldots, P-1)$ $\rightarrow$	Calculate the staged process for the controls $\hat{\mathbf{u}}^{[j]}(k)$ and set the state vectors $\mathbf{x}(k)$ equal to the state grid points $\mathbf{x}^{[j]}(k)$, $(k = 1, \ldots, P-1)$
UNTIL $j < N$ or a slave is not calculating	
Receive the state grid points $\leftarrow$	Send the state grid points $\mathbf{x}^{[j]}(k)$, $(k = 1, \ldots, P-1)$ to the master process
then $j < N$	
Increase j by one and start a non-calculating slave process for the calculation of the state grid points $\rightarrow$	Receive j and calculate the staged process with $\hat{\mathbf{u}}^{[j]}(k)$ and set the state vectors $\mathbf{x}(k)$ equal to the state grid points $\mathbf{x}^{[j]}(k)$, $(k = 1, \ldots, P-1)$
Send the state grid points $\mathbf{x}^{[j]}(k)$ $(j = 1, \ldots, N;\ k = 1, \ldots, P-1)$ to all slave processes $\rightarrow$	Receive the values and define the control grid points $\mathbf{u}^{[j]}(k)$ $(j = 1, \ldots, R;\ k = 1, \ldots, P)$ for the optimization

continued next page

Master Process	Slave Process
do $n = (P-1), 1$, step -1	
do $j = 1$, number of slave processes	
Start a slave process for the optimization of the stages $n+1$ to P and the initial vector $\mathbf{x}^{[j]}(n)$ $\longrightarrow$	$F_{\min} = \infty$ *For* $i = 1$ *to* R *step* 1 $\mathbf{x} = \mathbf{G}(\mathbf{x}^{[j]}(n), \mathbf{u}^{[i]}(n+1))$ $F = p_k(\mathbf{x}^{[j]}(n), \mathbf{u}^{[i]}(n+1))$ *For* $q = n+2$ *to* P *step* 1 Search $j^* := \{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(q- -1) - \mathbf{x})^2\}$ $F = F + p_q(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1)$ $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1))$ $F = F + g(\mathbf{x})$ $F < F_{\min}$ *then* $\mathbf{u}_{\text{opt}}^{[j]}(n+1) = \mathbf{u}^{[i]}(n+1)$ $F_{\min} = F$
UNTIL the stages $n+1$ to P are optimized for all state grid points $\mathbf{x}^{[j]}(n)$, $j = 1, \ldots, N$	
Receive and store the optimal controls $\mathbf{u}_{\text{opt}}^{[j]}(n+1)$ and the state vector $\mathbf{x}^{[j]}(n)$ at the entrance of stage $n+1$ $\longleftarrow$	Send the values of $\mathbf{u}_{\text{opt}}^{[j]}(n+1)$ to the master process
$j \leq N$ then	
Increase j by one and start the non-calculating slave process for the optimization of stages $n+1$ to P and the state vector $\mathbf{x}^{[j]}(n)$ $\longrightarrow$	$F_{\min} = \infty$ *For* $i = 1$ *to* R *step* 1 $\mathbf{x} = \mathbf{G}(\mathbf{x}^{[j]}(k-1), \mathbf{u}^{[i]}(k))$ $F = p_k(\mathbf{x}^{[j]}(k-1), \mathbf{u}^{[i]}(k))$ *For* $q = k+1$ *to* P *step* 1 Set $j^* := \{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(q- -1) - \mathbf{x})^2\}$ $F = F + p_q(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1))$ $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}_{\text{opt}}^{[j^*]}(q-1))$ $F = F + g(x)$ $F < F_{\min}$ then $\mathbf{u}_{\text{opt}}^{[j]}(k) = \mathbf{u}^{[1]}(k)$ $F_{\min} = F$

continued next page



```
Master Process                                              | Slave Process
        Send the optimal controls                           ──→ Receive the data
        $\mathbf{u}^{[j]}_{opt}(n+1),\ j = 1, \ldots, N$ to all
        slave processes
    $F_{min} = \infty$
    For $i = 1$ to $R$ step $1$
        $\mathbf{x} = \mathbf{G}(\mathbf{x}_0, \mathbf{u}^{[i]}(1)$
        $F = p_k(\mathbf{x}_0, \mathbf{u}^{[i]}(1)$
        For $q = 2$ to $P$ step $1$
            Set $j^* := \{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(q-1) - \mathbf{x})^2\}$
            $F = F + p_q(\mathbf{x}, \mathbf{u}^{[j^*]}_{opt}(q-1))$
            $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}^{[j^*]}_{opt}(q-1))$
        $F = F + g(\mathbf{x})$
        $F < F_{min}$
        then
            $\mathbf{u}^{[j]}_{opt}(k) = \mathbf{u}^{[i]}(k)$
            $F_{min} = F$
    $j^* = 1, F = 0, \mathbf{x} = \mathbf{x}_0$
    For $k = 1$ to $P$ step $1$
        $\mathbf{r}(k) = \gamma \mathbf{r}(k)$
        $\mathbf{x} := \mathbf{G}(\mathbf{x}, \mathbf{u}^{[j^*]}(k))$
        $F = F + p_k(\mathbf{x}, \mathbf{u}^{[j^*]}_{opt}(k))$
        $\mathbf{u}^*(k) = \mathbf{u}^{[j^*]}_{opt}(k)$
        $k < P$
        then
            Set $j^* := \{j = 1, \ldots, N \mid \min(\mathbf{x}^{[j]}(k) - \mathbf{x}^2)\}$
    $F = F + g(\mathbf{x})$
minimal value of the objective funtion: $F$,
optimal controls: $\mathbf{u}^*(k)$, $(k = 1, \ldots, P)$
```

Fig. 4. Flow diagram of the Parallel Iterative Dynamic Programming

The flow diagram of the parallel IDP algorithm is given in *Fig. 4.* This diagram is directly related to the algorithm in *Fig. 2.* The algorithm employs a master/slave scheme to distribute the tasks. The master process mainly coordinates the work of the slave processes which first generate in parallel the state grids and then the staged processes are optimized in parallel for each state grid point. To generate the state grids the staged process is calculated for different controls $\mathbf{u}(k)$, starting from the initial state $\mathbf{x}_0$. N trajectories will be calculated by which N grid points are generated in each state grid. Each of the trajectories can be calculated independently of the others in a slave process. After the generation of the state grid, the stages are to be optimized starting from the last stage P. The subproblems presented in *Eqs.* (4) and (5) are calculated one after the other. Each of the subproblems ($k = P$, ..., 2) is to be optimized for different state vectors $\mathbf{x}^{[j]}(k)$ of the state grid k. The optimization of a subproblem for each vector of the state grid can be done independently of the optimization of the subproblem of another state grid vector. Therefore, the optimization calculations can be distributed amongst the slaves. Each slave process calculates an optimal control which is transferred to the master process at the end of the calculation. Each of the slave processes is realized on a separate processor, for example, several personal computers that are connected by means of network interface cards. The more processors are available the more slave processes can be operated in parallel, and the more computing time can be saved, although there is some loss owing to data exchange and manipulation.

3. Example

In the present work the parallel IDP algorithm was used for the optimization of a problem given by LUUS (1993a). The algorithm was executed on a cluster of up to four HEWLETT PACKARD 735 workstations and on a PARSYTEC GC/PP 128 Parallel Computer where up to eleven PPC 601 processors were used. One master process running on one processor spawns a number of slave processes on each of the other processors.

The parallelization of the algorithm was measured by three commonly employed parameters defined as follows:

- The maximal speedup S_C is the ratio of the calculation time t_S of the fastest sequential algorithm and the calculation time t_C of the fastest parallel algorithm executed on C processors

$$S_C = \frac{t_S}{t_C}. \tag{7}$$

- The algorithm speedup $\overline{S}_C$ is the ratio of the calculation time t_1 of the parallel algorithm executed on one processor and the calculation time t_C of the parallel algorithm executed on C processors

$$\overline{S}_C = \frac{t_1}{t_C}. \tag{8}$$

- The efficiency E_C is the ratio of the algorithm speedup S_C and the number of processors C

$$E_C = 100\frac{\overline{S}_c}{C}. \tag{9}$$

The calculation time of the sequential algorithm was 6360 [s] on a HP 735 workstation und 10019 [s] on a single PPC 601 processor. In *Tables 1* and *2* the computation times t_C of the parallel algorithm, the maximal speedup S_C, the algorithm speedup $\overline{S}_C$ and the efficiency E_C are presented for up to 11 parallel processors.

Table 1. HP Workstation - Cluster: Calculation time (t_C), maximal speedup (S_C), algorithm speedup ($\overline{S}_C$) and efficiency (E_C) as a function of the number of processors (C), sequential time is 6360 [s] (N=7 state grid points and M=7 control grid points)

C	t_C [s]	S_C	$\overline{S}_C$	E_C [%]
1	6749	0.94	1.00	100
2	3807	1.67	1.77	88.5
3	3118	2.04	2.16	72.0
4	2520	2.52	2.69	67.0

Table 2. Parsytec GC/PP 128: Calculation time (t_C), maximal speedup (S_C), algorithm speedup ($\overline{S}_C$) and efficiency (E_C) as a function of the number of processors (C), sequential time is 10019 [s] (N=11 state grid points and M=11 control grid points)

C	t_C [s]	S_C	$\overline{S}_C$	E_C [%]
1	10507	0.95	1.00	100.0
2	5933	1.69	1.77	88.5
3	4286	2.34	2.45	81.7
4	3442	2.91	3.05	76.3
5	3249	3.08	3.23	64.6
6	2647	3.79	3.97	66.2
7	2582	3.88	4.07	58.1
8	2538	3.95	4.14	51.8
9	2510	3.99	4.19	46.6
10	2483	4.04	4.23	42.3
11	1966	5.10	5.34	48.5

The maximal speedup S_C for one processor ($C = 1$) was 0.94 on the HP-Cluster and 0.95 on the Parsytec (see *Tables 1, 2*). Therefore 5–6% of the calculation time of the parallel algorithm is needed for the communication of slave and master process. If the slave processes are optimally synchronized the speedup increases linearly with the number of processors which is shown in *Figs 5* and *6*.

The difference in the communication overhead is based on the different data transport layers (see *Tables 1, 2*, EC). The HP-Cluster runs on a Fiber Distributed Data Interface (FDDI). FDDI is a 100-MBit/sec token-passing ring that uses optical fiber for transmission between stations and has dual counter-rotating rings to provide redundant data paths for reliability. The Parsytec GC/PP 128 uses a 2D grid of separate communication processors each connected with two PPC 601 processors.

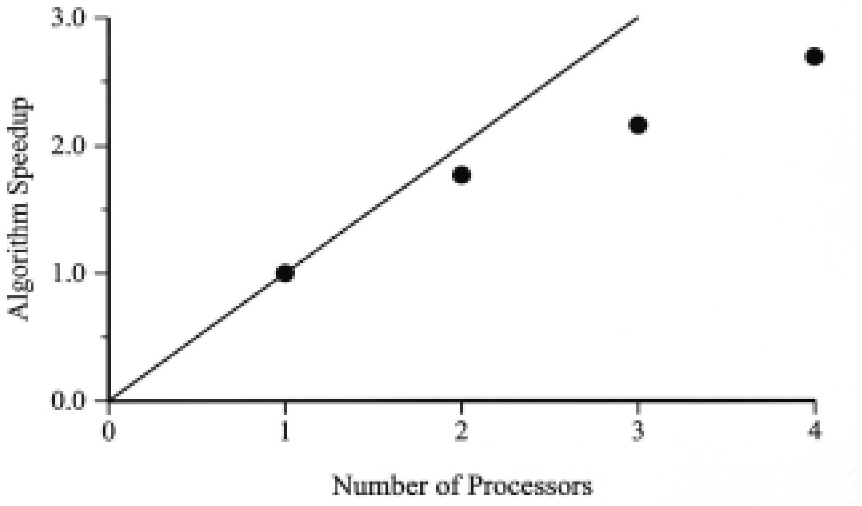


Fig. 5. Algorithm speedup as a function of the number of processors *C* (HP Workstation – Cluster)

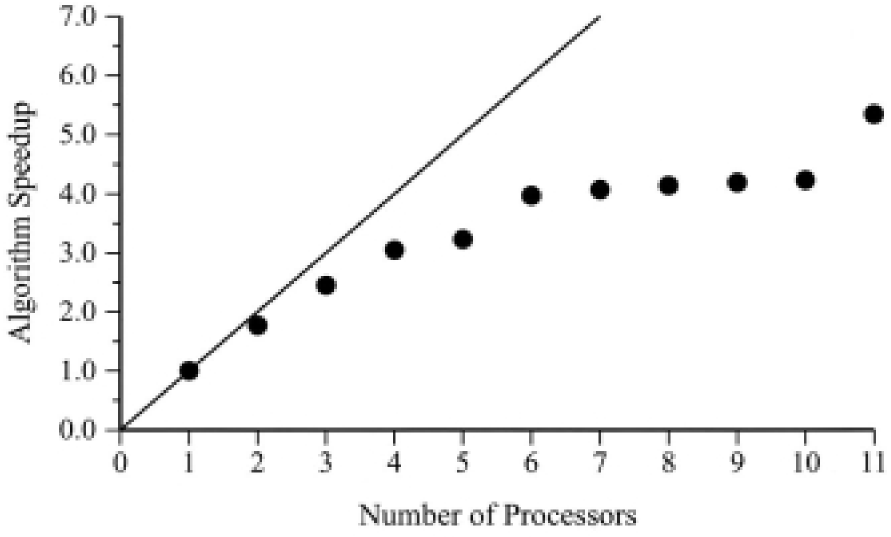


Fig. 6. Algorithm speedup as a function of the number of processors *C* (Parsytec GC/PP 128)

This strategy allows a very high network bandwidth and by far faster data exchange than the FDDI network which is limited to 100 MBit/sec.

Another important point is the latency time which represents the time to create a message. On network based PVM implementations this time is very long because of the administrative work of the operating system. The Parsytec GC/PP 128 uses the special operating system PARIX which reduces the latency time. The Parsytec implementation of PVM is based on these PARIX calls so that the spawning of messages is very fast.

Because of the simple implementation the synchronisation, and therefore the efficiency E_C, depend on the ratio of the number of grid points and the number of processes (*Tables 1, 2*). If the number of grid points is a multiple of the number of processors and the calculation time of the staged process is always the same, the generation of the state grid points, and the optimization are finished at the same time by all slave processes and, consequently, no slave process is waiting and reducing the efficiency of the parallelization. In this work IDP was executed with $N = 7$ state grid points and $M = 7$ control grid points. If the algorithm runs with 2 processors, the first 6 sets of state grid points are generated in parallel. The seventh set is generated by only one slave process while the second slave process is waiting. For three processors, the efficiency is even more reduced, since at the end two slave processors are waiting. Therefore, the number of state grid points and the number of control grid points should be a multiple of the number of processors in order to get a high efficiency of the parallelization.

Nomenclature

C	number of parallel slave processes
E_C	efficiency of the parallel algorithm
F	objective function of the staged process
F_k	function defined by *Eqs* (5) and (6)
G	function for the state vector
g	objective function of the state vector $\mathbf{x}(P)$
P	number of stages
N	number of grid points in each state grid
$p(k)$	value of the objective function of stage k
p_k	objective function of stage k
R	number of grid points in each control grid
$r(k)$	radius of the search region of control $\mathbf{u}(k)$
$\underline{S}_C$	maximal speedup
$\overline{S}_C$	algorithm speedup
t_S	calculation time of the sequential algorithm [s]
t_C	calculation time of the parallel algorithm with C processors [s]
$\mathbf{u}(k)$	control vector of stage k
$\mathbf{u}^*(k)$	midpoint of the search region of control $\mathbf{u}(k)$
$\mathbf{u}^{[j]}(k)$	j-th point of control grid k
$\hat{\mathbf{u}}^{[j]}(k)$	j-th point of control grid k for calculating the state grid point $\mathbf{x}^{[j]}(k)$
$\mathbf{u}_{opt}^{[j]}(k)$	optimal control vector for the j-th point of state grid k
$\mathbf{x}_0$	initial state vector of the process
$\mathbf{x}(k)$	state vector at the exit of stage k and at the entrance of stage $k+1$
$\mathbf{x}^{[j]}(k)$	j-th point of state grid k

Greek Symbols

α (k)	lower boundary of control $\mathbf{u}(k)$
β (k)	upper boundary of control $\mathbf{u}(k)$
γ	reduction factor

References

[1] BELLMAN, R. E. – DREYFUS, S. E. (1962): Applied Dynamic Programming, Princeton University Press, Princeton, NJ.

[2] BERTSEKAS, D. P. – TSITSIKLIS, J. N. (1989): Parallel and Distributed Computation, Numerical Methods, Prentice Hall, Englewood Cliffs.

[3] BOJKOV, B. – LUUS, R. (1992): Use of Random Admissible Value for Control in Iterative Dynamic Programming. *Ind. Eng. Chem. Res.*, Vol. 31, p. 1308.

[4] CHEN, M. C. (1986): Design Methodology for Synthesizing Parallel Algorithms and Architectures, *J. of Parallel and Distributed Computing*, Vol. 3, p. 461.

[5] CLEMENTI, E. – CHIN, S. – CORONGIU, G. – DETRICH, J. H. – DEPUIS, M. – FOLSOM, D. – LIE, G. C. – LOGAN, D. – SOMAND, V. (1989): Supercomputing and Supercomputers for Science and Engineering in General and for Chemistry and Biosciences in Particular, *Int. J. Quant. Chem.*, Vol. 35, p. 3.

[6] EDMONDS, P. – CHU, E. – GEORG, A. (1993): Dynamic Programming on Shared-Memory-Multiprocessor Parallel Computing, Vol. 19, p. 9.

[7] GEIST, A. – BEGUELIN, A. – DONGARRA, J. – JIANG, W. – MANCHEK, R. – SUNDERAM, V. (1994): PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge.

[8] GEIST, A. – BEGUELIN, A. – DONGARRA, J. – JIANG, W. – MANCHEK, R. – SUNDERAM, V. (1993): PVM 3.0 User's Guide and Reference Manual, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

[9] GROPP, W. – LUSK, E. – SKJELLUM, A. (1994): Using MPI – Portable Parallel Programming with the Message-Passing Interface, The MIT Press, Cambridge, MA.

[10] HARTIG, F. – KEIL, F. J. – LUUS, R. (1995): Comparison of Optimization Methods for a Fed-Batch Reactor, *Hung. J. Ind. Chem.*, Vol. 23, p. 141.

[11] HARTIG, F. – KEIL, F. J. (1993a): A Modified Algorithm of Iterative Dynamic Programming, *Hung. J. Ind. Chem.*, Vol. 21, p. 101.

[12] HARTIG, F. – KEIL, F. J. (1993b): Large Scale Spherical Fixed Bed Reactors: Modeling and Optimization, *Ind. Eng. Chem. Res.*, Vol. 32, p. 424.

[13] JAJA, J. (1992): An Introduction to Parallel Algorithms; Addison Wesley, Reading, Mass.

[14] LUUS, R. (1999): Iterative Dynamic Programming, CRC Press (in preparation).

[15] LUUS, R. (1993a): Optimization of Fed-Batch Fermentors by Iterative Dynamic Programming, *Biotechnol. Bioeng.*, Vol. 41, p. 599.

[16] LUUS, R. (1993b): Application of Dynamic Programming to Differential-Algebraic Process Systems, *Computer Chem. Engng.*, Vol. 17, p. 373.

[17] LUUS, R. (1990): Application of Dynamic Programming to High-Dimensional Non-Linear Optimal Control Problems, *Int. J. Control*, Vol. 52, p. 239.

[18] LUUS, R. (1989): Optimal Control by Dynamic Programming Using Accessible Grid Points and Region Reduction, *Hung. J. Ind. Chem.*, Vol. 17, p. 523.

[19] LUUS, R. – GALLI, M. (1991): Multiplicity of Solutions in Using Dynamic Programming for Optimal Control, *Hung. J. Ind. Chem.*, Vol. 19, p. 55.

[20] LUUS, R. – ROSEN, O. (1991): Application of Dynamic Programming of Final State Constrained Optimal Control Problems, *Ind. Eng. Chem. Res.*, Vol. 30, p. 1525.

[21] MCBRIAN, O. A. (1994): An Overview of Message Passing Environments, *Parallel Computing*, Vol. 20, p.417.

[22] MCRAE, G. J. (1990): Chemical Process Modeling and Simulation Using Advanced Computational Architectures, In: Foundation of Computer-Aided Process Design; Siirola, J.J., Grossmann, I.E., Stephanopoulos, G. (Eds.), Elsevier, New York.

[23] RYTTER, W. (1988): On Efficient Parallel Computations for Some Dynamic Programming Problems, *Theoret. Comput. Sci.*, Vol. 58, p. 257.

[24] SMITH, J.; (1993): The Design and Analysis of Parallel Algorithms, Oxford University Press, Oxford.

[25] SUNDERAM, V. S. – GEIST, G. A. – DONGARRA, J. – MANCHEK, R. (1994): The PVM Concurrent Computing System: Evolution, Experiences and Trends, *Parallel Computing*, Vol. 20, pp. 531–545.

[26] WILSON, G. V. (1995): Practical Parallel Programming; MIT Press, Cambridge, MA.