A Skeleton for Parallel Dynamic Programming

D. Morales, F. Almeida, F. Garcia, J. Gonzalez, J. Roda, and C. Rodriguez

Centro Superior de Informática
Dpto. E.I.O. y Computación
Universidad de La Laguna, Tenerife, Spain
falmeida@ull.es

Abstract. The development of skeleton tools constitutes an alternative to cover the gap between current parallel architectures and sequential programmers. Its contruction involves formal models, paradigms and methologies. Based in the automata theory we have developed a formal model for Parallel Dynamic Programming over pipeline networks. This model makes up a paradigm which is the core of skeleton tools oriented to the Dynamic Programming Technique. Following the methodology coerced by the model, we present a tool that provides the user with the ability to obtain parallel programs adapted to the parallel architecture. The efficiency is contrasted on three current parallel platforms: Cray T3E, IBM SP2 and SG Origin 2000.

1 Introduction

Dynamic programming (DP) is an important problem-solving technique that has widely been used in various fields such as control theory, operations research, biology and computer science. Sequential computation for dynamic programming has been studied extensively. General discrete DP programs viewed as multistage finite automatons with a certain superimposed cost structure were presented in [7]. Parallel dynamic programming algorithms for specific problems and specific recurrence relations have been presented for different architectures [2, 4, 8, 6]. However, few effort has been done for a general parallel dynamic programming formal model. The only proposal for a general parallel DP approach was due to Wah et al. in 1985 [10]. They suggest parallelizations of certain classes of DP formulations using parallel matrix product algorithms. However, the resolution of DP problems through matrix products leads, in general, to inefficient solutions. Neither this approach nor any other, lead to a general tool or skeleton for DP.

Following Ibaraki's [7] discrete DP approach, we extend the sequential model for DP to a parallel model. We propose a general parallel dynamic programming algorithm for pipeline and ring networks for multistage automatons. The parallel algorithm presented constitutes a paradigm to solve DP problems and provides the conceptual framework for the development of skeleton tools. Based in this paradigm, we present a skeletton oriented tool for the implementation of Multistage DP algorithms. The level of this skeleton tool not only simplifies the writing of parallel programs but also the development of the sequential case.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 877–887, 1999.

The tool presented derives a parallel DP program from the user specification of a sequential code. The tool manages the optimal mapping of the stages of the associated DP automata in the actual processors, the generation of the corresponding messages and the handle of the involved data structures.

To study the performance of our skeleton, experiments were carried out in three different architectures: a distributed memory computer, the IBM-SP2; a distributed-shared memory computer, the Origin 2000 and a distributed memory with shared address space computer, the Cray T3E. The SP2 from IBM has 44 processors, 12 GB main memory, 431 GB disk storage and 18.55 Gflop/s theoretical peak. The Origin 2000 from Silicon Graphics is a 64 R10000 processors (196 MHz) machine with 8 GB main memory, 288 GB disk and 25.08 Gflop/s theoretical peak. The Cray T3E has 32 DEC 211164 processors. Each processor has 128 Mb of memory and reaches 600 Mflops.

Section 2 introduces a formal approach for DP based in the automata theory. The model is illustrated with the Single Resource Allocation Problem. Section 3 presents the class of Non Decreasing Finite Automata, for which an optimal pipeline algorithm is delivered. In section 4, we describe tools for the automatic parallelization of DP. In section 5, we will show the computational results. Finally, section 6 presents the conclusions.

2 The Dynamic Programming Technique

Dynamic Programming is an important technique for solving combinatorial optimization problems. It is based on the principle of optimality introduced by R. Bellman [3]. Our goal is to solve the optimization problem:

$$\begin{array}{c} minimize\ (maximize)\ f(x) \\ subject\ x\ \epsilon\ S \end{array} \tag{1}$$

in which the set of constraints S can be modeled as a language, that is, $S \subseteq \Sigma^*$. The set Σ is a finite set called the set of decisions. The elements of Σ^* which are elements of the Kleene closure of Σ, are referred or called policies. A policy $x \epsilon \Sigma^*$ is said to be feasible if $x \epsilon S$.

The Dynamic Programming technique works by associating to the optimization problem an automaton with costs [7]:

Definition: An automaton with costs is a triple $\prod = (M,\ \mu,\ \xi_0)$, where:

1) M is a finite automaton $M = (Q,\ \Sigma,\ q_0,\ \lambda,\ Q_F)$ in which:
   Q denotes the set of states,
   Σ is the set of decisions,
   $q_0 \epsilon Q$ is the initial state,
   $\lambda : Q \times \Sigma \rightarrow Q$, is the transition function and
   $Q_F \subset Q$ is the set of final states.

The transition function (λ) can be extended to the set of policies; $\lambda : Q \times \Sigma^* \rightarrow Q$ by defining $\lambda(q, \epsilon) = q$ and $\lambda(q, xa) = \lambda(\lambda(q, x), a)$, where $q \epsilon Q$,

$x \in \Sigma^*$, and $a \in \Sigma$. The notation $\lambda(x) \cong \lambda(q_0, x)$ for all $x \in \Sigma^*$ and $\lambda(q) = \{q' \in Q \text{ such that exists } a \in \Sigma : \lambda(q,a) = q'\}$ will be used.
2) $\mu : \Re \times Q \times \Sigma \to \Re$ is the cost function, where $\Re$ denotes the set of real numbers. If decision $a \in \Sigma$ is applied to state $q \in Q$, and the accumulated cost is ξ, the automaton changes into the new state $\lambda(q,a)$ with new cost given by $\mu(\xi, q, a)$. The cost function μ can be extended to $\Re \times Q \times \Sigma^* \to \Re$ by defining $\mu(\xi, q, xa) = \mu(\mu(\xi, q, x), \lambda(q, x), a)$ for $\xi \in \Re$, $q \in Q$, $x \in \Sigma^*$, $a \in \Sigma$ and $\mu(\xi, q, \epsilon) = \xi$. We will use notation $\mu(x)$ to denote the cost of policy x starting in q_0, i.e.: $\mu(x) \cong \mu(\xi_0, q_0, x)$.
3) $\xi_0 \in \Re$ is the setup cost of the initial state q_0.

Definition: A finite automaton with costs $\prod = (M, \mu, \xi_0)$ represents the optimization problem (1) if and only if it satisfies:
1) The language accepted by the automaton M is the set S of feasible solutions, i.e.: $L(M) = \{x \in \Sigma^* \ / \ \lambda(x) \in Q_F\} = S$.
2) For all feasible policy $x \in S$, $\mu(x) = f(x)$.

Definition: A feasible policy $x \in S$ is said to be optimal if and only if $\mu(x) \leq \mu(y)$ for any other feasible policy $y \in S$.

Definition: For any state $q \in Q$ we define $G(q)$ as the optimal value of any policy leading from the start state q_0 to state q, i.e., $G(q) \cong \min\{\mu(x) \ / \ x \in \Sigma^*, \lambda(x) = q\}$ when there exists an $x \in \Sigma^*$ such that $\lambda(x) = q$, and $G(q) = +\infty$ otherwise.

Let be G^* the optimum of the values for the feasible policies, that is $G^* = \min\{G(q) / \ q \in Q_F\}$. To solve the optimization problem (1) is equivalent [7] to finding the optimal value G^*, i. e., $G^* = \min\{f(x) \ / \ x \in S\}$. The optimal value $G(q)$ can be obtained from the optimal values of the states from wich q is reacheable through some decision a, i.e.,

for all $q \neq q_0$, $G(q) = \min\{\mu(G(q'), q', a) \ / \ q' \in Q, a \in \Sigma, \lambda(q', a) = q\}$ and $G(q_0) = \xi_0$.

Definition: Let M be a finite automaton, M is a multistage automaton if and only if, there is a partition of the set Q, $Q = \{q_0\} \cup Q_1 \cup ... \cup Q_n \cup Q_F \cup \{q_d\}$ such that $Q_i \cap Q_j = \emptyset$ when $i \neq j$, $q_d \in Q - Q_F$ is called the dead state, and
$\lambda(q_0, a) \in Q_1 \cup \{q_d\}$, for all $a \in \Sigma$
$\lambda(q, a) \in Q_{i+1} \cup \{q_d\}$, for all $q \in Q_i$, a $\in \Sigma$, $i = 1, 2, .., n-1$.
$\lambda(q_d, a) = q_d$, for all $a \in \Sigma$.
$\lambda(q, a) \in Q_F \cup \{q_d\}$, for all $q \in Q_n$ and for all $a \in \Sigma$.

The sequential dynamic programming algorithm in figure 1 solves the optimization problem (1) for multistage automatons with costs. The algorithm assumes the existence a FIFO queue where the values needed to compute the next STAGE will be stored. In figure 1, functions INSERT and REMOVE introduces and eliminates data from the queue respectively. The algorithm runs

in time $O(\mid Q \mid \cdot \mid \sum \mid \cdot O(\mu))$. Following this methodology, our tool will create to the user the illusion of being writting and executing sequential programs while the code generated will be parallel code. The user omits the loop in line 2 of figure 1 and the tool will automatically substitute the calls to INSERT and REMOVE by the classical message passing primitives to send and receive data. In the parallel case, the queue will disappear at all.

```
1:Begin
2:   for STAGE = 0 to n do
3:   begin
4:     if (STAGE == 0) /* First STAGE */
5:     begin
6:         G(q_0) = ξ_0;
7:         INSERT(G(q_0));
8:     end
9:     else /* General STAGE */
10:        for q_{STAGE-1 i} ∈ Q_{STAGE-1} do
11:        begin
12:          REMOVE (G(q_{STAGE-1 i}));
13:          for a ∈ Σ such that λ(q_{STAGE-1 i}, a) = q_{STAGE i} do
14:            G(q_{STAGE i}) = min{G(q_{STAGE i}), μ(G(q_{STAGE-1 i}), q_{STAGE-1 i}, a)};
15:          if (STAGE != n) /* last STAGE do not INSERT in queue */
16:            INSERT (G(q_{STAGE i}));
17:          for q_{STAGE j} = λ(q_{STAGE-1 i}, a) with a ∈ Σ do
18:            G(q_{STAGE j}) = min{G(q_{STAGE j}), μ(G(q_{STAGE-1 i}), q_{STAGE-1 i}, a)};
19:        end;
20:  end;
21:End.
```

Fig. 1. Sequential Dynamic Programming Algorithm

2.1 An Example: The Single Resource Allocation Problem (RAP)

The Single Resource Allocation Problem can be stated as follows [7]:

$$\begin{array}{c} maximize\ z\ = \sum_{j=1}^{N} f_j(x_j) \\ subject\ to \sum_{j=1}^{N} x_j = M \end{array} \tag{2}$$

Namely, it is required to allocate M units of an indivisible resource to N activities so that the sum of the effectiveness measured by $f_j(x_j)$ is maximized.

In order to solve this problem by dynamic programming, we introduce the following states q_{km} and their values $G(.)$.

q_{00}: The initial state representing the null allocation.

q_{km}: The state representing k-dimensional allocation vectors $(x_1, x_2, ..., x_k)$ of nonnegative integers such that $\sum_{j=1}^{k} x_j = m$, for $k = 1, 2, ..., N$, $m = 0, 1, ..., M$.

q_{NM}: The final state representing the allocation vectors satisfying (2).

$G(q)$: The maximum of $\sum_{j=1}^{k} f_j(x_j)$ for the allocation vectors represented by state q.

The transition function can be defined as: $\lambda(q_{km}, x) = q_{k+1,m+x}$ $x = 0, ..., M - m$

By definition, $G(q_{NM})$ is the optimum objective value we want to compute. It is equal to the length of a longest path from the source q_{00} to the sink q_{NM} where each edge (q_{k-1m}, q_{km+x_k}) has a length $f_k(x_k)$. The value $G(q_{km})$ can be recursively computed by

$G(q_{00}) = 0$

$G(q_{1m}) = f_1(m), m = 0, ..., M$

$G(q_{km}) = \max_{0 \le x_k \le m}\{G(q_{k-1\ m-x_k)} + f_k(x_k)\}, k = 2, 3, ..., N, m = 0, 1, ..., M.$

3 Parallelization of the Dynamic Programming Technique

Definition: A multistage automaton M is called a non decreasing automaton if, and only if, there is a total order for each stage, $Q_k = \{q_{k1}, ..., q_{km}\}$, such that for every q_{ki} on stage Q_k, the condition $\lambda(q_{ki}) \subset [q_{k+1i}, q_{k+1r}]$, where $r = |Q_{k+1}|$ is satisfied.

```
1:Begin
2:   for q_{k-1i} ϵ Q_{k-1} do
3:   begin
4:     ReceiveFromLeft (G(q_{k-1i}));
5:     for a ϵ ∑ such that λ(q_{k-1i}, a) = q_{ki} do
6:         G(q_{ki}) = min{G(q_{ki}), μ(G(q_{k-1i}), q_{k-1i}, a)};
7:     SendToRight (G(q_{ki}));
8:     for q_{kj} = λ(q_{k-1i}, a) with a ϵ ∑ do
9:         G(q_{kj}) = min{G(q_{kj}), μ(G(q_{k-1i}), q_{k-1i}, a)};
10:  end;
11:END
```

Fig. 2. A Pipeline Algorithm Parallelizing on the Stages for non decreasing automatons (PAPS). Code for processor k.

Along the rest of this paragraph we will consider the automaton M in $\prod = (M, \mu, \xi_0)$ to be a non decreasing automaton.

Since we are dealing with a multistage automaton, the problem can be solved using a pipeline with as many processors as stages. The computation of optimal values in stage k, $G(q)$, $q \in Q_k$ is allocated to processor k (Algorithm PAPS of

figure 2). During the *ith* iteration processor k receives from processor $k-1$ the optimal value $G(q_{k-1})$. As soon as $G(q_{k-1})$ is received, the values $G(q)$ for those q such that there exists $a \epsilon \Sigma$ with $\lambda(q_{k-1}, a) = q$ are updated. Property 1 of a non decreasing automaton grants that all the optimal values from which state q_{ki} depends on have been received, and so the optimal value $G(q_{ki})$ is completely computed and can be sent to processor $k+1$. The algorithm stops when processor n computes the optimal values for the final states $q_f \epsilon Q_F$. Processor n holds the optimal solution G^*. The conditions for the optimality of this algorithm can be found in [1].

4 Tools for Parallel Dynamic Programming

We aim to the building of an skeleton for parallel DP based on the exposed methodology. The project is divided in two stages: 1) The construction of a tool (called *La Laguna Pipeline*, *llp*) providing the programmer with a virtually infinite pipeline. The programmer just has to take control of the behavior of a general cell of the pipe. *llp* simulates the total amount of processors and maps them onto the physical ones using a mix of block and cyclic policies adapted to the grain g of the target architecture. 2) The developement of the parallel dynamic programming skeleton (called *La Laguna Parallel DP* or *llpdp*), built on top of the *llp*.

```
void solver_1 (int N) {
  int  result;
  if (NAME == 0) f_0(); /* First stage */
  else if (NAME == (N - 1)) {
    result = f_n-1(); /* Last stage */
    GPRINTF("\n% d:Result = % d",NAME,result);
  }
  else f(); /* A general stage of the pipeline */
  BROADCAST(BCAST, "variable", count, offset);
}
```

Fig. 3. First and Last Processors run different code.

4.1 The Basic Tool: Giving Support for Pipeline Paralellism

La Laguna Pipeline, *llp*, is a general purpose tool for the pipeline programming paradigm, and it is not specifically conceived for DP. *llp* is conceived as a macro based library and its portability is guaranteed by the wide number of message passing libraries supported (MPI, PVM, InmosC). Figures 3, 4 and 5 illustrate the easiness of use of applying a general pipeline scheme. In figure 3 funcion solver_1(), constitutes a virtually infinite pipeline to be executed. The

code for the first and last processors of the pipe must be differentiated from the remaining processors. The variable NAME identifies the virtual processor in the pipeline. The functions f_i can use the macros IN and OUT to establish the corresponding communications with the left and right neighbors. llp, provides parallel Input/Output routines as the $GPRINTF$ call in figure 3, allowing processors to read and write from shared input and output streams. The macro $BROADCAST$ update values of global data distributed around the processors previously declared as $SHARED$ (line 8 in figure 4). As in figure 4, several pipelines can be iteratively executed. A call to the macro $PIPE$ has as arguments the generic routine and the size of the pipeline, $solver\_1()$ and N respectively for the first case. This macro introduces the routine in a loop simulating the virtual processors assigned to the actual processor according to the mapping policy.

```
1: #include "llp.h"
2: ....
3: int  main (PARAMETERS) {
4:   INITPIPE;
5:   Type variable[MAX];
6:   if (ARGC != 3)  abort();
7:   N = atoi(ARGV(1)); M = atoi(ARGV(2));
8:   SHARED(&variable, "variable", Type);
9:   PIPE(solver_1(N), N);
10:  PIPE(solver_2(M), M);
11:  EXITPIPE;
12: }
```

Fig. 4. Function main(): expanding PIPEs of N and M virtual processors.

4.2 Implementing Cyclic and Blocking-Cyclic Mapping Policies

The algorithm in figure 2, requires a variable number of processors, usually proportional to the number of stages or the number of states per stage. It does not consider the fact that for most cases the number of processors required exceeds the number of available processors. Therefore, it is necessary to assign the processes between the available processors. This is the problem of finding an efficient mapping of the virtual pipeline on the actual parallel machine. Fortes and Moldovan propose in [9] to partition the set of processes in sets $B(i)$ of the same cardinal as the number p of processors, in such a way that process $q \in B(i)$ if and only if $q/p = i$. Process q is processed by processor $q \ mod \ p$. All processes in the same set $B(i)$ are processed in parallel. Implementation can be easily achieved on a one way ring topology where the first and last processors are connected through a buffering process. This approach leads to a pure cyclic mapping. Better results depending on the grain g of the underlying platform

may be achieved if this cyclic mapping is combined with block mapping. On a block-cyclic mapping, process q is assigned to processor (q/g) *mod* p. The implementation of a pure cyclic mapping policy for a pipeline can be easily obtained over a ring as a particular case of the macro PIPE described in figure 5 taking $g = 1$. Every processor calculates the number of STAGES to compute, (n/p) (line 2), and then goes into a loop running the function f for the appropiated first virtual processor NAME (line 4).
The classical technique to obtain block-cyclic mapping consisting of g sequential executions of the f function is not a good approximation to the problem. The delay introduced by each processor produces a parallel algorithm as slow as the sequential one. To obtain an efficient implementation, processors must start to work as soon as possible and they must be feeded with data when needed. To fulfil this conditions we implement the block-cyclic mapping using the Unix standard library "*setjmp.h*". This library allows unconditional jumps to variable labels. To execute function f in pipeline with grain g, automatically g copies of the local data of function f are expanded, one for each one of the processes to be executed in one iteration of the loop. The code to be executed is the same for each process: the function f. Therefore, only the Program Counter of every process need to be stored. Macros IN and OUT differentiate between internal and external communications. External comunications use the communication primitives provided by PVM/MPI. Internal comunications produce the context switch between processes using the functions *setjmp*() and *longjmp*() of the library "*setjmp.h*". The context switch just consist of updating the NAME of the process to get the Program Counter and the address of the local variables for this process. Since a context switch occurs only when needed and it is implemented very efficiently, the overhead introduced by the tool in a block-cyclic partition is minimum.

```
1: #define PIPE(f, n) {
2:     int bands = numbands(n, g); /* The number of STAGES to compute */
3:     for (i = 0; i < bands; i++) {
         /* Physical Processor pname calculates the */
         /* NAME of the first process to simulate */
4:       NAME = g * (pname + i * numproc);
5:       if (NAME ≤ (n - 1)) f;
6:     }
7: }
```

Fig. 5. The Macro PIPE.

4.3 The Dynamic Programming Skeleton, llpdp

The *llpdp* dynamic programming skeleton takes advantage of the fact that the behavior of a pipeline is similar to a FIFO queue. This way, the *llpdp* user is asked

to write the sequential code for a generic automaton stage using a FIFO queue provided by the *llpdp* system. Instead of speaking of processor NAMEs, the *llpdp* uses STAGES and IN and OUTS are substituted by INSERTs and REMOVEs. This simply renaming and semantic change means that the code generated is similar that the generated by the *llp* tool. The parallel code generated is optimal when the automaton is Non Decreasing and the user inserts the states following the natural automaton order. The efficiency achieved by the skeleton is similar to the obtained by the *llp* programming system. The *llpdp* code in figure 6 expresses the DP formulation of the Single Resource Allocation problem. Note the similarity of this code with the sequential one of figure 1.

```
void   solve_RAP () {
  LOCAL_VAR;
     int m, j, x, G[M + 1];
  BEGIN;
  if (STAGE == 0) /* First STAGE */
     for( m = 0; m ≤ M; m++) {
       G[m] = 0;
       INSERT(G[m]);
     }
  else { /* General STAGE */
       for( m = 0; m ≤ M; m++) G[m] = 0;
       for (m = 0; m ≤ M; m++) {
         REMOVE(&x);
         G[m] = max(G[m], x + f(STAGE-1, 0));
         if (STAGE != N) INSERT(G[m], 1, sizeof(int));
         for (j = m + 1; j ≤ M; j++) G[j] = max(G[j], x + f(STAGE - 1, j - m));
       } /* for m ... */
       } /* else ... */
  END;
} /* solve_RAP */
```

Fig. 6. Simple Pipeline Algorithm for the RAP. Every processor runs the same Code.

5 Computational Results

Table 1 shows the times of the sequential algorithm in the three machines. Table 2 presents the times of the algorithm implemented using the tool. The executions were performed varying the grain from 1 to 20 and the number of processors ranging from 2 to 8. The results obtained for each machine appear in two columns. The column labeled $Time$ indicates the time in seconds, while the labeled S keeps the speedup. The introduction of mechanisms to manage the grain implies

a slight loss of performance. It can be detected comparing the results with grain 1 and grain 2. However, an increasing of the grain generally carries an increase of the performance.

Table 1. Sequential time. Problem of size N = 400, M = 1000

Cray T3E	IBM SP2	Origin 2000
27.46	88.29	28.47

Table 2. Times and speedups for the RAP. Problem of size N = 400, M = 1000

		Cray T3E		IBM SP2		Origin 2000	
Grain	Proc.	Time	S	Time	S	Time	S
1	2	27.03	1.0	70.49	1.3	35.50	0.8
1	4	14.00	2.0	36,71	2.4	19.59	1.5
1	8	7.51	3.7	18.63	4.7	16.53	1.7
2	2	36.63	0.7	69.87	1.3	44.60	0.6
2	4	18.56	1.5	35.05	2.5	22.35	1.3
2	8	9.47	2.9	17.71	5.0	11.22	2.5
5	2	36.06	0.8	68.13	1.3	43.07	0.7
5	4	18.26	1.5	34.24	2.6	21.57	1.3
5	8	9.33	2.9	17.20	5.1	10.87	2.6
10	2	36.24	0.8	67.72	1.3	42.87	0.7
10	4	18.31	1.5	33.95	2.6	21.47	1.3
10	8	9.25	3.0	17.07	5.2	10.77	2.6
20	2	36.16	0.8	67.52	1.3	42.59	0.7
20	4	18.25	1.5	33.86	2.6	21.35	1.3
20	8	7.41	3.7	13.62	6.5	8.57	3.3

6 Conclusions

A general procedure that can be applied to the whole class of dynamic programming problems have been presented. The proposed scheme leads to efficient implementations of both parallel and sequential dynamic programs, as confirm the computational experiments presented in this paper. Based in the proposed algorithms, a general parallel tool for Dynamic Programming has been built. This tool comes to fill a remarkable absence of such a kind of tool both in the sequential and parallel fields.

Acknowledgments

We wish to thank to CEPBA, CESCA and CIEMAT for allowing us the access to their machines.

References

[1] Almeida F.,Models in Parallel Dynamic Programming for the Discrete Case. PHDegree. DEIOC. Unniversidad de La Laguna. October 1996.
[2] Andonov, Raimbault, Quinton. *Dynamic Programming Parallel Implementations for the Knapsack Problem*. Technical Report 740. IRISA. June 1993.
[3] Bellman R.. *Dynamic Programming*. Princeton U. P.. 1957.
[4] Chen G., Jang J.. *An Improved Parallel Algorithm for 0/1 Knapsack Problems*. Parallel Computing. 18. 811-821. 1992.
[5] Edmonds P., Chu E., George A.. *Dynamic Programming on a Shared Memory Multiprocessor*. Parallel Computing. 19, 9-22. 1993.
[6] Gibbons A., Rytter W., *Efficient Parallel Algorithms*. Cambridge University Press. 1988.
[7] Ibaraki T.. *Enumerative Approaches to Combinatorial Optimization, Part II*. Annals of Operations Research. Volume 11, N. 1-4, 1988.
[8] Kindervater G., Trienekens H.. *Experiments with Parallel Algorithms for Combinatorial Problems*. European Journal of Operational Research. 33, 65-81. 1988.
[9] Moldovan D., Fortes J.. *Partitioning and Mapping Algorithms into fixed size Systolic arrays*. IEEE Trans. Comput. C-35 (1), 1-12. 1986.
[10] Wah B., Li G., Fen C.. *Multiprocessing of Combinatorial Search Problems*. Computer. Vol. 18. 6, 93-108. 1985.