A Tool for Parallel Dynamic Programming Algorithms

D. González, F. Almeida, J. Roda, C. Rodríguez

Centro Superior de Informática

Dpto. E.I.O. y Computación

Universidad de La Laguna, Tenerife, Spain

casiano@ull.es

Abstract. Dynamic Programming is an important paradigm that has been widely used to solve problems in various areas as control theory, operation research, biology and computer science. We generalize the Finite Automaton Formal Model for Dynamic Programming deriving pipeline parallel algorithms. The optimality of these algorithms is established for the new class of Non Decreasing Finite Automata. As an intermediate step for the construction of skeleton for the automatic parallelization of Dynamic Programming, we have developed a tool for the implementation of pipeline algorithms. The tool maps the processes in the pipeline in the target architecture following a mix of block and cyclic policies adapted to the grain of the machine. Based on the former tool, the automatic parallelization of Dynamic Programming is straightforward. The use of the model and its associated tools is illustrated with the Single Resource Allocation Problem. The performance and portability of these tools is compared with specific "hand made" code written by experi-

enced programmers. The experimental results on distributed memory, shared distributed memory and shared memory architectures prove the scalability of the proposed paradigm and its associated tools.

Keywords Models, Tools, Dynamic Programming, Resource Allocation

1 Introduction

Dynamic programming (DP) is an important problem-solving technique that has been widely 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 Dynamic Programming programs viewed as multistage finite automatons with a certain cost structure superimposed were presented in [12, 15].

Many authors describe parallel algorithms for specific Dynamic Programming problems. Parallel dynamic programming algorithms for 0/1, two-dimensional and integer knapsack problems have been proposed for different architectures in [1, 3, 4, 6, 16, 17, 20, 23, 25].

Specific recurrence relations such as shortest paths, optimal binary search tree, optimal chained matrix multiplication parenthesizations, etc. have been approached for systolic arrays [18, 22], general shared-memory multiprocessors [8] and for PRAM models [9, 10]. All these contributions were proposed for specific problems and specific architectures. The only proposal for a general parallel DP approach was due to Wah et al. in 1985 [26]. They suggest parallelizations in terms of a classification of the dynamic programming problems according to the

form of the functional equation. The general idea of their work is that solving a dynamic programming problem can be reduced to finding the shortest paths in the associated graph. This problem can be solved by the iterative multiplication of matrices. From a practical point of view these matrix algorithms present, for the majority of cases, an unnecessary and prohibitive complexity: $O(n^3 logn)$, where n is the number of vertex in the graph. Neither this approach nor any other, lead to a general tool or skeleton for DP. It is a remarkable fact the absence of such kind of tools, undoubtedly due to the absence of general methods

Following Ibaraki's [12] discrete Dynamic Programming approach, we extend the sequential model for DP to a parallel model. We propose general parallel dynamic programming algorithms for pipeline and ring networks for multistage automatons. The study of the optimality (using the definition given in [14]) of these algorithms lead us to the introduction a new class of multistage automatons: Non Decreasing Automatons. Many Dynamic Programming problems fit in the class of multistage automaton described [2, 3, 20, 1, 4]. The parallel algorithm presented constitute a paradigm to solve DP problems. Paradigms are specially important in Parallel Computing since they allow to consider separately the problems and to think in a clear and methodical mode in the opportunities to parallelize, abstracting the details of the particular problem that make an arduous task the work of the programmer.

As an intermediate step for the automatic parallelization of Dynamic Programming, we have developed a tool for the implementation of general pipeline algorithms. The tool maps the processes in the pipeline in the target architec-

ture following a mix of block and cyclic policies optimized in the grain of the machine. Based on the former tool, the automatic parallelization of Dynamic Programming is straightforward.

To study the performance of the proposed approach, 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 shared memory machine, the DEC AlphaServer 8400. We had access to the machines at Centre Europeu de Paral-lelisme de Barcelona. The SP2 from IBM has 12 + 32 processors, 12 GB main memory, 431 GB disk storage and 18.55 Gflop/s theoretical peak. The AlphaServer 8400 (DEC) has 10 processors Alpha 21164 (440 Mz), 2 GB main memory and 60 GB disk storage and a peak performance of 8,80 Gflop/s. The Origin 2000 from Silicon Graphics is a 32 + 32 R10000 processors (196 MHz) machine with 8 GB main memory, 288 GB disk and 25.08 Gflop/s theoretical peak.

Section 2 introduces the Finite Automaton Formal Model for Dynamic Programming. The model is illustrated with the Single Resource Allocation Problem. A general sequential algorithm is derived from the model. Section 3 presents the class of Non Decreasing Finite Automata, for which an optimal pipeline algorithm is delivered. Tools for automatic parallelization of DP will be described in section 4. The efficiency of these tools is compared with specific "hand made" code written by experienced programmers. The experimental results for the Single Resource Allocation Problem prove the scalability of the proposed paradigm and its associated tools in these three significant parallel architectures.

2 The Dynamic Programming Technique

Dynamic Programming is an important technique for solving combinatorial optimization problems. It is based in the principle of optimality introduced by R. Bellman [5]. 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 the Kleene closure of Σ, Σ^* are called policies. A policy $x\ \epsilon\ \Sigma^*$ is said feasible if $x\ \epsilon\ S$.

The Dynamic Programming technique works by associating to the optimization problem an automaton with costs [12]:

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

1) M is a finite automaton $M = (Q,\ \Sigma,\ q_0,\ \lambda,\ Q_F)$ [11].

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 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 $q\ \epsilon\ Q$, $x\ \epsilon\ \Sigma^*$, and $a\ \epsilon\ \Sigma$. The notation $\lambda(x) \cong \lambda(q_0,\ x)$ for all $x\ \epsilon\ \Sigma^*$ will be used. There is no risk

of confusion between the two uses of λ, since the number and the type of the arguments allow the reader to differentiate them. The same consideration holds for the following notation: $\lambda(q) = \{q' \epsilon\ Q\ /\ \exists\ a\ \epsilon\ \Sigma :\ \lambda(q,a) = q'\}$.

2) $\mu : \Re \times Q \times \Sigma \rightarrow \Re$ is the cost function, where $\Re$ denotes the set of real numbers. If decision a $\epsilon\ \Sigma$ is applied to state $q\ \epsilon\ Q$, 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 $\Re\ \times Q\ \times \Sigma^* \rightarrow \Re$ by defining $\mu(\xi, q, xa) = \mu(\mu(\xi, q, x),\ \lambda(q, x), a)$ for $\xi\ \epsilon\ \Re$, $q\ \epsilon\ Q$, $x\ \epsilon\ \Sigma^*$, $a\ \epsilon\ \Sigma$ and $\mu(\xi, q, \epsilon) = \xi$.

3) $\xi_0\ \epsilon\ \Re$ is the setup cost of the initial state q_0.

We will use notation $\mu(x)$ to denote the cost of policy x starting in q_0, i.e.: $\mu(x) \cong \mu(\xi_0, q_0, x)$ and $\rho(q)$ denotes the cost of the best policy starting at q, $\rho(q) = \min\{\mu(G(q), q, x)\ /\ \lambda(q,x)\epsilon\ Q_F,\ x\ \epsilon\ \Sigma^*\}$. We will also define $d^-(q) = |\lambda^{-1}(q)|$ and $d^+(q) = |\lambda(q)|$.

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.: $L(M) = \{x\ \epsilon\ \Sigma^*\ \ /\ \lambda(x)\ \epsilon\ Q_F\} = S$.

2) For all feasible policy $x\ \epsilon\ S$, $\mu(x) = f(x)$.

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

Along this work we will assume that function μ is monotone in ξ and taking positive values $\mu(\xi, q, x) \geq 0$.

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

Let be $G^* = \min\{G(q)/ q \epsilon Q_F\}$. Obviously, to solve the optimization problem (1) is equivalent to find the optimal value G^* :

Theorem 2.1: If $\prod = (M, \mu, \xi_0)$ represents problem (1) then, $G^* = \min\{f(x) / x \epsilon S\}$.

proof: See [12].

The following recursive formula is the base for the Dynamic Programming Algorithm:

Theorem 2.2: For all $q \neq q_0$,

$$G(q) = \min\{\mu(G(q'), q', a) / q' \epsilon Q, a \epsilon \Sigma, \lambda(q', a) = q\}$$

and $G(q_0) = \xi_0$.

proof: See [12].

Definition: Let be M 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 $i \neq j$, and

$\lambda(q_0, a) \epsilon Q_1 \cup \{q_d\}$, for all $a \epsilon \Sigma$

$\lambda(q, a) \epsilon Q_{i+1} \cup \{q_d\}$, for all $q \epsilon Q_i$, a $\epsilon \Sigma$, $i = 1, 2, .., n-1$.

$\lambda(q_d, a) = q_d$, for all $a \epsilon \Sigma$.

$\lambda(q, a) \epsilon Q_F \cup \{q_d\}$, for all $q \epsilon Q_n$ and for all $a \epsilon \Sigma$, where $q_d \epsilon Q - Q_F$ is called the dead state.

We will denote by $\Sigma(q)$ the set $\{b \epsilon \Sigma \ / \ \lambda(q, b) \neq q_d\}$.

The sequential dynamic programming algorithm in figure 1 straightforwardly solves the optimization problem (1) for multistage automatons with costs. Note that to obtain $G(q)$ we are assuming that it is necessary to compute in run time the states q' on which it depends.

Theorem 2.3: The complexity of algorithm in figure 1 is $O(\mid Q \mid \cdot \mid \sum \mid \cdot O(\mu))$.

proof: Trivial.

Through this work we will assume a constant time for the computation of function μ. For those cases not keeping this hypothesis the complexities can be obtained by simply multiplying by the factor $O(\mu)$ the computational times presented along the paper.

2.1 An example: the Single Resource Alloaction Problem (RAP)

The Single Resource Allocation Problem can be stated as follows [2, 13]:

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

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

```
1:Begin
2:   G(q_0) = ξ_0;
3:   for k = 1 to n do
4:   begin
5:     for q ∈ Q_k do
6:       G(q) = ∞;
7:     for a ∈ Σ do
8:       for q' ∈ Q_{k-1}
9:       begin
10:        q = λ(q', a);
11:        G(q) = min{G(q), μ(G(q'), q', a}};
12:      end;
13:  end;
14:End.
```

Fig. 1. Sequential Dynamic Programming Algorithm.

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 length $f_k(x_k)$. The value $G(q_{km})$ can be recursively computed by

$G(q_{00}) = 0$

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

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

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 three following conditions are satisfied:

1) $\lambda(q_{ki}) \subset [q_{k+1i}, q_{k+1r}]$, where $r = |Q_{k+1}|$

2) $|\lambda(q_{ki})| \leq |\lambda(q_{k+1i})|$

3) There is a constant C such that $|\{a \epsilon \Sigma \ \lambda(q_{k-1i}, a) = q_{ki}\}| \leq C$.

Note that the condition $|\lambda(q_{ki})| \leq |\lambda(q_{k+1i})|$ is equivalent to $d^+(q_{ki}) \leq d^+(q_{k+1i})$.

Among the examples of non decreasing automatons we can mention the DP formulations of the 0-1, integer and non linear knapsack problems [20, 21] resource allocation problems [13] and some scheduling problems [24].

Lemma 3.1: $|Q_k| \leq |Q_{k+1}|$ for any nondecreasing automaton M.

proof: Is a trivial consequence of property 1) of a non decreasing automaton.

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

Since we are dealing with a multistage automaton we can exploit the property that, for all $q \in Q_k$ the set $\{q' / \lambda(q', a) = q\}$ is included in Q_{k-1}. This fact allows to solve the problem using a pipeline with as many processors as stages by allocating the computation of optimal values in stage k, $G(q)$, $q \in Q_k$, to processor k (Algorithm PAPS of figure 2). In many cases, processor k has enough information about the stage $k-1$ to build the states of Q_{k-1} in constant time. That is, it knows the number of states in the previous stage $|Q_{k-1}|$ and the order in which they will be computed and received. This order has to be the natural order given in Q_{k-1} by the non decreasing automaton. 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 exist $a \in \Sigma(q_{k-1})$ 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^*.

1:begin

2: for $q_{k-1i} \epsilon Q_{k-1}$ do

3: begin

4: ReceiveFromLeft $(G(q_{k-1i}))$;

5: for $a \epsilon \sum$ such that $\lambda(q_{k-1i}, a) = q_{ki}$ do

6: $G(q_{ki}) = \min\{G(q_{ki}),\ \mu(G(q_{k-1i}),\ q_{k-1i},\ a)\}$;

7: SendToRight $(G(q_{ki}))$;

8: for $q_{kj} = \lambda(q_{k-1i},\ a)$ with $a \epsilon \sum(q_{k-1i})$ do

9: $G(q_{kj}) = \min\{G(q_{kj}),\ \mu(G(q_{k-1i}),\ q_{k-1i},\ a)\}$;

10: end;

11:end.

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

The goal of the rest of this paragraph is to establish the conditions to guarantee an optimal speedup for the proposed algorithm under the former assumption.

Lemma 3.2: In the algorithm PAPS, processor n, in charge of the last stage, starts to work at iteration $O(n)$.

proof: Obviously, the first processor sends the first optimal value in time $O(d^-(q_{11})) = O(1)$. By induction, processor k receives at iteration $O(k)$ the first optimal

value $G(q_{k-1\ 1})$. State q_{k1} is completely updated after the completation of the $O(d^-(q_{k1}))$ iterations of the loop in lines 5-6. Condition 3) of the non decreasing automaton definition implies $O(d^-(q_{k1})) = O(1)$. Thus, processor $k+1$ receive the first value $G(q_{k1})$ in time $O(k)$.

In many cases the special characteristics of the considered problem allow a clever strategy leading to an implementation of the loop 8-9 with complexity $O(|\lambda(q_{k-1i})|)$.

Lemma 3.3: Processor k of the pipeline spends just a constant time at line 4 of the algorithm PAPS waiting for values of the left neighbor $k-1$.

proof: The lemma is trivial for the first processor. Let us assume that processor $k-1$ waits a constant time for the values of its left neighbor and let us see that processor k waits a constant time. After sending the value $G(q_{k-1i})$ (to avoid delays in the output of line 7 we assume, if necessary, the presence of buffers) processor $k-1$ goes into the loop of line 8-9. This loop has a complexity minor than $O(|\lambda(q_{k-2i})|)$. Meanwhile, processor k is traversing the two loops of lines 5-6 and 8-9 plus the sending of line 7. This leads to a complexity $O(|\lambda(q_{k-1i})|)$. Since M is a non decreasing automaton $|\lambda(q_{k-2i})| \leq |\lambda(q_{k-1i})|$, and so processor $k-1$ sends the optimal value $G(q_{k-1i+1})$ to processor k after completing the loop of lines 5-6 having complexity $O(1)$. Thus the lemma holds.

Theorem 3.1: Let $\prod = (M, \mu, \xi_0)$ be a non decreasing automaton with costs and let be $m = |Q_n|$, then the algorithm PAPS has complexity: $O(n + \sum_{i=1}^{m} d^-(q_{ni}))$.

proof: As a consequence of the lemma 3.2, processor n starts to work on time $O(n)$ and since according to lemma 3.3 no delay due to communication occurs, then it spends $O(\sum_{i=1}^{m} d^-(q_{ni}))$ with the computation.

Theorem 3.2: Let $\prod = (M, \mu, \xi_0)$ be a non decreasing automaton with costs such that exists a constant D holding the condition:

$$D \sum_{k=1}^{n} \sum_{q_{ki} \epsilon Q_k} d^-(q_{ki}) = n \sum_{i=1}^{m} d^-(q_{ni})$$

under this assumption the algorithm PAPS is optimal.

proof: It is a direct consequence of theorem 3.1, theorem 2.3 and the fact that the hypothesis of the theorem guarantees an optimal load balancing.

The condition of the theorem 3.2 holds if all the stages have the same cardinal.

4 Tools for Parallel Dynamic Programming

Most of the efficient parallel techniques for solving DP [1, 2, 3, 4, 6, 20] exploit pipeline parallelism. This kind of algorithms, as the algorithm in figure 2 require a variable number of processors, usually proportional to the number of stages or the number of states per stage. They do 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 among 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 [22] 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 the 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 $(q/g) \ mod \ p$. We aim to the building of an skeleton for parallel dynamic programming based in the methodology exposed in the former paragraph. Three are the main objectives: Easiness of use, Portability and Efficiency. To achieve these goals the project is divided in two stages: the first one is the construction of a tool (called $LaLagunaPipeline, llp$) providing the programmer with a virtually infinite pipeline. The programmer just has to take control of the behaviour of a general cell of the pipe. The *La Laguna Pipeline* 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. The second one is properly the construction of the parallel dynamic programming skeleton or frame (called *La Laguna Parallel DP* or *llpdp*), built in top of the *llp*.

4.1 The basic tool: Giving Support for Pipeline Paralellism

La Laguna Pipeline is a general purpose tool for the pipeline programming paradigm, and it is not specifically conceived for Dynamic Programming. The portability of the tool is guaranteed by the wide number of message passing libraries supported (MPI, PVM, InmosC). The intensive use of macros instead

of subroutines contributes to diminish the overhead introduced by the tool and enlarges its efficiency, as will be proved in the computational results in this section. Figures 3, 4 and 5 illustrates the easiness of use applying the general pipeline scheme presented in figure 2 to the Single Resource Allocation Problem (RAP) described in section 2.1.

```
void solve_RAP (void) {
  int  result;
  if (NAME == 0)
    ini_stage();
  else if (NAME == (N - 1)) {
    result = last_stage();
    GPRINTF("\ n % d:Result = % d",NAME,result);
  }
  else
    transition();
}
```

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

The call to the macro $PIPE$ in figure 5 has as arguments the generic routine, $solve_RAP()$ and the size of the pipeline N. This macro introduces the $solve_RAP()$ routine in a loop simulating the virtual processors assigned to the

```
void   transition ()
{
  int m, j, x;
  for( m = 0; m ¡= M; m++)
     G[m] = 0;
  for (m = 0; m ¡= M; m++) {
     IN(&x);
     G[m] = max(G[m], x + f(NAME-1, 0));
     OUT(&G[m], 1, sizeof(int));
     for (j = m + 1; j ¡= M; j++)
       G[j] = max(G[j], x + f(NAME - 1, j - m));
  } /* for m ... */
} /* transition */
```

Fig. 4. Simple Pipeline Algorithm for the RAP. Code for Processor NAME.

actual processor according to the mapping policy. The variable $NAME$ identifies the virtual processor in the pipeline. Macros IN and OUT establish the corresponding communications with the left and right neighbors. La Laguna Pipeline provides parallel Input/Output routines as the $GPRINTF$ call in figure 3, allowing processors to read and write from shared input and output streams.

Table 1 compares the performance of *llp* against the same algorithm written by an experienced programmer using the message passing library PVM (subcol-

```
int  main (PARAMETERS)
{ INITPIPE;
  if (ARGC != 3) {
     abort();
  }
  N = atoi(ARGV(1)); /* Number of activities */
  M = atoi(ARGV(2)); /* Units of resource */
  PIPE(solve_RAP(), N);
  EXITPIPE;
}
```

Fig. 5. Function main(): expanding a PIPE of N virtual processors.

umn labeled ad-hoc). Columns are labeled with the three considered architectures and rows with the number of processors. As can be expected, unless for one entry, the ad-hoc column presents better times than the corresponding llp times. However, the loss of efficiency $LE = (adhoc - llp)/ad - hoc$ is only 6in the IBM-SP2. In the Origin there is an anomalous super-efficiency for 8 processors. The loss of efficiency for 2 processors is 4.5tests on the Digital were executed sharing the machine with other users. While the user code still scales for 8 processors, the tool time starts to increase. Table 2 presents the speedup of the llp. Except for the Digital, an acceptable scalability is observed.

Table 1. llpdp versus ad-hoc. Times in seconds for the RAP. Problem of size N = 400, M = 1000

	IBM-SP2		O2000		DEC Alphaserver	
Proc.	ad-hoc	llp	ad-hoc	llp	ad-hoc	llp
Seq.	210.46	-	28.48	-	17.47	-
2	148.01	156.41	29.54	30.87	13.37	15.52
4	74.72	79.11	15.50	15.68	6.82	8.00
8	37.81	40.22	9.53	8.16	6.77	8.50

Table 2. Speedup for the RAP using llpdp. Problem of size N = 400, M = 1000

Proc.	IBM-SP2	O2000	DEC Alphaserver
2	1.35	0.92	1.13
4	2.66	1.82	2.20
8	5.23	3.49	2.00

4.2 The Dynamic Programming Skeletton

Sequential dynamic programming algorithms (figure 1) follow a bread-first-search in which first stages of the automaton are visited in first place. It is well known that this kind of traversing implies the use of a FIFO queue. The *llpdp* dynamic programming skeleton takes advantage of the fact that a user can not distinguish the behaviour of the former pipeline from 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 exactly the same that the generated by the llp tool for the code in figure 4. The parallel code generated is optimal when the automaton is Non Decreasing and the user inserts the states following the natural automaton order. Obviously, the efficiency achieved by the tools is equal to the obtained by the *llp* programming system.

5 Conclusions

We give a general parallel dynamic programming procedure that can be applied to a wide range of problems. The proposed scheme admit an easy and efficient implementation for the class of multistage automaton described as confirm the computational experiments in [2, 3, 20]. These computational ex periments agree with the predicted theoretical results. The proposed algorithms constitute a suitable kernel to build general parallel tools for Dynamic Programming like the described in the former section.

Acknowledgments

We wish to thank to the Centre Europeu de Paral-lelisme de Barcelona and the Centre de Supercomputacio de Catalunya for allowing us the access to their machines.

References

1. Aleksandrov V.. *Dynamic Programming Approach of the Knapsack Problems for the Ring of Processors.* WPDP93. Sofia. Bulgaria. 1993.
2. Almeida F., Morales D., Roda J.L., Rodríguez C., García F.. *The Single Resource Allocation Problem: Parallel Algorithms on Distributed Systems.* Proceedings of the 19th Technical Conference of the World Occam and Transputer User Group (WOTUG). Nottingham. 1996.
3. Almeida F., García F., Morales D., Rodríguez C.. *A parallel algorithm for the integer knapsack problem for pipeline networks.* Journal of Parallel Algorithms and Applications 6 (3/4). 1994.
4. Andonov, Raimbault, Quinton. *Dynamic Programming Parallel Implementations for the Knapsack Problem.* Technical Report 740. IRISA. June 1993.
5. Bellman R.. *Dynamic Programming.* Princeton U. P.. 1957.
6. Chen G., Jang J.. *An Improved Parallel Algorithm for 0/1 Knapsack Problems.* Parallel Computing. 18. 811-821. 1992.
7. Burns A., Friedman F.. *Parallel Ordered Enumerative Algorithms for the Multidimensional Knapsack Problem on Transputer Networks.* Proceedings of the Seventh Conference of the North American Transputer Users Group (NATUG 7). Atlanta. 116-132. IOS Press. 1995.
8. Edmonds P., Chu E., George A.. *Dynamic Programming on a Shared Memory Multiprocessor.* Parallel Computing. 19, 9-22. 1993.
9. Galil Z., Park K..*Parallel Algorithms for Dynamic Programming Recurrences with more than O(1) Dependency.* Journal of the Parallel and Distributed Computing. 21, 213-222. 1984.
10. Gibbons A., Rytter W., *Efficient Parallel Algorithms.* Cambridge University Press.

1988.

11. Hopcroft J., Ullman J.. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company. 1939.

12. Ibaraki T.. *Enumerative Approaches to Combinatorial Optimization, Part II.* Annals of Operations Research. Volume 11, N. 1-4, 1988.

13. Ibaraki T., Katoh N.. *Resource Allocation Problems. Algorithmic Approaches.* The MIT Press. Cambridge, Massachusetts. London. 1988.

14. JáJá J. *An Introduction to Parallel Algorithms.* Addison Wesley. 1992.

15. Karp R., Held M.. *Finite-State Processes and Dynamic Programming.* SIAM J. Appl. Math. 15, 693-768. 1967.

16. Kindervater G., Trienekens H.. *Experiments with Parallel Algorithms for Combinatorial Problems.* European Journal of Operational Research. 33, 65-81. 1988.

17. Lee J., Shragowitz E., Sahni S.. *A hypercube algorithm for the 0/1 knapsack problem.* Journal of Parallel and Distributed Computing 5, pp. 438-456.

18. Louka B., Tchuente M.. *Dynamic Programming on 2D Systolic Arrays.* Parallel Processing. 265-275. 1988.

19. Moldovan D., Fortes J.. *Partitioning and Mapping Algorithms into fixed size Systolic arrays.* IEEE Trans. Comput. C-35 (1), 1-12. 1986.

20. Morales D., Roda J., Almeida F., Rodríguez C., García F.. *Integral Knapsack Problems: Parallel Algorithms and their Implementations on Distributed Systems.* Proceedings of the 1995 International Conference on Supercomputing. 218-226. ACM Press. 1995.

21. Morin T., Marsten R.. *Branch-and-Bound Strategies for Dynamic Programming.* Operations Research. Vol. 24. 4, 611-627. 1976.

22. Myoupo J.. *Mapping Dynamic Programming onto Modular Linear Systolic Arrays.*

Distributed Computing. 6, 165-179. 1993.

23. Teng S.. *Adaptative parallel algorithms for integral Knapsack Problems. Journal of Parallel and Distributed Computing.* 8, 400-406. 1990.

24. Syslo M., Deo N., Kowalik J.. *Discrete Optimization Algorithms with Pascal Programs.* Prentice-Hall. Eglewood Cliffs. New Jersey. 1983.

25. Ulm D., Wang P.. *Solving a two-dimensional Knapsack Problem on SIMD Computers.* International Conference on Parallel Processing. 1992.

26. Wah B., Li G., Fen C.. *Multiprocessing of Combinatorial Search Problems.* Computer. Vol. 18. 6, 93-108. 1985.