PARALLEL DYNAMIC PROGRAMMING ALGORITHMS

Marinus Veldhorst

RUU-CS-85-27
October 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

PARALLEL DYNAMIC PROGRAMMING ALGORITHMS

Marinus Veldhorst

Technical Report RUU-CS-85-27
October 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

PARALLEL DYNAMIC PROGRAMMING ALGORITMS

Marinus Veldhorst

Department of Computer Science, University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht, The Netherlands.

ABSTRACT

In this paper we will present a number of parallel algorithms for the dynamic programming problem

$$c(i,i) = 0 \quad (0 \leq i \leq n)$$
$$c(i,j) = w(i,j) + \min_{i<m\leq j} (c(i,m-1)+c(m,j))$$

A standard example is the construction of optimal binary search trees (cf. [5]). Sequential algorithms run in $O(n^3)$ time or, if the quadrangle inequality holds (cf. [8]), in $O(n^2)$ time. For the former we will design parallel algorithms that run in $O(n^3/p)$ time on $p<n^2$ processors. We will also show that dynamic programming problems satisfying the quadrangle inequality can be solved in $O(n^2/p + n \log \log p)$ time using p $(1<p\leq n)$ processors. A global shared memory is assumed.
Moreover, we will design a systolic array for computing the $c(i,j)$'s that runs in linear time using $\Theta(n^2)$ processors.

1. Introduction. Many combinatorial optimization problems can be solved with the dynamic programming technique. Essential in it is the principle of optimality (cf. [3]): the optimal solution can be written as a recurrence relation in optimal solutions of subproblems. Computed optimal solutions (and possibly additional information) of subproblems are maintained in memory so that they are computed only once. This latter aspect creates also the difficulty of dynamic programming: storage requirements may become unacceptably high (the "curse of

dimensionality" (cf. [3])).

In this paper we will consider the dynamic programming problem

Compute c(0,n) defined as

$$c(i,i) = 0 \quad (0 \le i \le n)$$
$$c(i,j) = w(i,j) + \min_{i<m\le j} (c(i,m-1)+c(m,j)) \quad \text{for } 0 \le i < j \le n \qquad (1)$$

in which the values w(i,j) ($0 \le i \le j \le n$) are known.

A straightforward sequential algorithm to solve (1) is given in algorithm A. It runs in time $O(n^3)$.

Example 1. Optimal binary search trees (cf. [5]). Given n keys a_1, ..., a_n and 2n+1 probabilities p_1, ..., p_n and q_0, ..., q_n where

p_i = probability that a_i is the search argument ($1 \le i \le n$)

q_i = probability that the search argument is between a_i and a_{i+1}.

The problem is to find a binary search tree which minimizes the expected number of comparisons during a search. Thus

$$\sum_{j=1}^{n} p_j * (1 + \text{level of } a_j) + \sum_{k=0}^{n} q_k * \left[\begin{array}{l} \text{level of external node that} \\ \text{corresponds to interval } (a_k, a_{k+1}) \end{array} \right]$$

must be minimized (the root of a binary tree has level 0).

Let c(i,j) be the cost (i.e., the expected number of comparisons during

```
(1)   for i:=0 to n do  c(i,i) := 0;
(2)   for k:=1 to n do begin
(3)      for i:=0 to n-k do begin
(4)         value := ∞;
(5)         for m:=i+1 to i+k  do
(6)             value := min(value, c(i,m-1)+c(m,i+k));
(7)         c(i,i+k) := value + w(i,i+k)
(8)      enddo
(9)   enddo
```

algorithm A: program to solve dynamic programming problem (1).

a search) of the optimal subtree T_{ij} with weights (probabilities for internal and external nodes) $p_{i+1}, \ldots, p_j$ and $q_i, \ldots, q_j$. Then

$$c(i,i) = 0 \quad (0 \le i \le n)$$
$$w(i,j) = p_{i+1} + \ldots + p_j + q_i + \ldots + q_j$$
$$c(i,j) = w(i,j) + \min_{i<m\le j} (c(i,m-1) + c(m,j))$$

and algorithm A can be used.

In many applications (e.g. example 1) algorithm A can be accelerated to run in $O(n^2)$ time, namely when the cost function c(i,j) satisfies the quadrangle inequality.

DEFINITION 1(cf. [8]). A function f(i,j) satisfies the quadrangle inequality if

$$f(i,j) + f(i',j') \le f(i',j) + f(i,j') \qquad (2)$$

for all $i \le i' \le j \le j'$ for which it is defined.

THEOREM 1(cf. [8]). If the cost function c in (1) satisfies the quadrangle inequality, then (1) can be solved in $O(n^2)$ time.

The acceleration is due to the fact that the possible values m for which the right hand side of (1) attains it minimum, is restricted. Let

$$R_c(i,j) = \max\{ m : c(i,j) = w(i,j) + c(i,m-1) + c(m,j)\}$$

Then, if c(i,j) satisfies the quadrangle inequality (2),

$$R_c(i,j-1) \le R_c(i,j) \le R_c(i+1,j)$$

Moreover, if w(i,j) is monotone increasing on the lattice of intervals and w(i,j) satisfies the quadrangle inequality, then c(i,j) satisfies the quadrangle inequality. Algorithm B gives a precise sequential version of this acceleration.

```
for i:=0 to n do begin  c(i,i) := 0; R(i,i) := i enddo;
for k:=1 to n do begin
   for i:=0 to n-k do begin
      value := ∞;
      for m:=R(i,i+k-1) to R(i+1,i+k)  do
         if value > c(i,m-1)+c(m,i+k))
         then  value := c(i,m-1)+c(m,i+k);  R(i,i+k) := m  endif;
      c(i,i+m) := value + w(i,i+m)
   enddo
enddo
```

algorithm B: program to solve a dynamic programming problem (1) that satisfies the quadrangle inequality.

In order to be able to assess parallel algorithms we will use the so-called speed-up and efficiency of processor utilization. Let X be an algorithm using p processors to solve some problem. Then the speed-up is defined as:

$$S_p(X) = \frac{\text{Time used by the most efficient sequential algorithm}}{\text{Time used by algorithm X on p processors}}$$

and the efficiency as:

$$E_p(X) = S_p(X)/p$$

Obviously we have $S_p(X) \leq p$ and $0 \leq E_p(X) \leq 1$. It is our purpose to design parallel algorithms for p processors that have a speed-up near p and an efficiency near 1.

In dynamic programming one distinguishes between stages, state variables and control variables and similarly between a stage loop, a state loop and a control loop that are nested (cf. [3]). In algorithm A these loops consist of the lines 2-9, 3-8 and 5-6, respectively. The term inspection is used in this paper to denote the execution of the body of the control loop for some value of m. Observe that in algorithm A the number of inspections in the control loop is independent of the state (i.e., the value of i). This is not the case in algorithm B.

In the past results on parallel algorithms for dynamic programming

problems used one of the following 3 ways (cf. [4], [1]):

(i) divide the control loop among the processors. Thus at each time there is one state such that all processors are performing inspections for this state.

(ii) divide the state loop among the processors. Each processor performs all inspections necessary for the state assigned to it. At any moment all processors (some of which may be idle) are performing inspections in the same stage.

(iii) divide the state loop and control loops among the processors. In [4] and [1] it is required that the number of controls is equal for all states. If this would not be the case some processors may be idle in their approach. At any moment all processors (some of which may be idle) are performing inspections for states in the same stage.

Also strategy (ii) may lead to inefficient use of processors when the number of inspections varies per state. We will prove that this may yield for dynamic programming problems satisfying the quadrangle inequality a speed-up that is constant even when an arbitrary number of processors can be used.

This paper is organized as follows. In section 2 we will consider algorithms A and B as a task system with some precedence constraints. In section 3 we will design a linear time systolic implementation of algorithm A using $O(n^2)$ processors. In section 4 we will prove that each parallel algorithm following strategy (ii) for solving (1) must run in $\Omega(n^2)$ time on any number of processors even when the problem satisfies the quadrangle inequality. In this section we will also present $O(n^3/p)$ time algorithms that use $p(\leq n^2)$ processors. In section 5 we will present a parallel algorithm to solve dynamic programming problems with the quadrangle inequality in $O(n \log \log n)$ time on a shared memory computer with n processors. We will generalize this result for shared memory computers with p $(1<p\leq n)$ processors.

2. A precedence graph. We consider algorithms A and B to consist of a number of tasks J_{ij} $(0 \leq i \leq j \leq n)$ such that J_{ij} computes the cost $c(i,j)$. Because the task J_{ij} uses a number of other costs $c(i',j')$, we can define a precedence relation on the set $J_n = \{J_{ij} : 0 \leq i \leq j \leq n\}$

$$J_{ij} << J_{i',j'} \quad \text{if} \quad (i=i' \text{ or } j=j')$$

Moreover define the relation $<$ as

$$J_{ij} < J_{i',j'} \quad \text{if} \quad (i=i' \text{ and } j=j'-1) \text{ or } (i=i'+1 \text{ and } j=j')$$

The relation $<<$ includes the relation $<$ and moreover, if $J_{ij} << J_{i'j'}$, then there are $i=i_1,\ldots,i_k=i'$ and $j=j_1,\ldots,j_k=j'$ such that.

$$J_{i_1 j_1} < J_{i_2 j_2} < \ldots < J_{i_k j_k}$$

For our purposes it suffices to use the relation $<$ instead of $<<$. Figure 1 shows graphically the relation $<$ in the form of a pyramid. In this pyramid, stage k consists of tasks $J_{m,m+k}$ $(0 \leq m \leq n-k)$, the j^{th} left

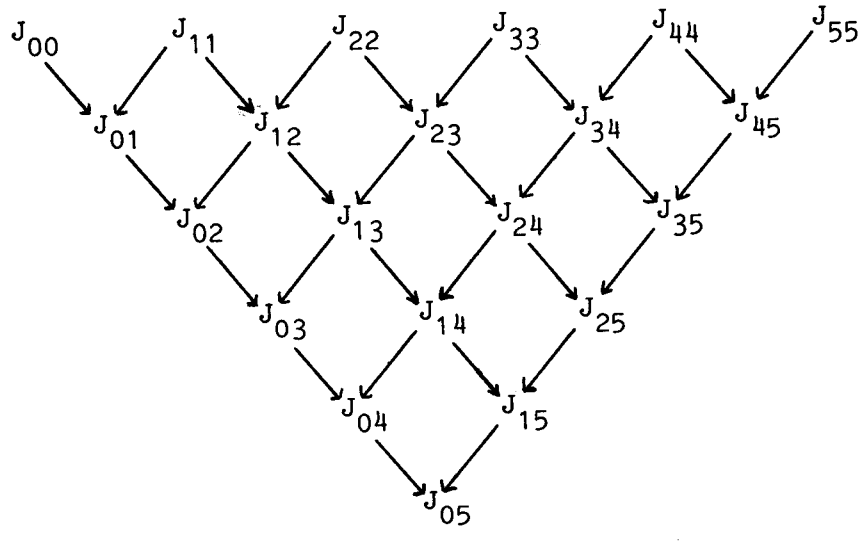

Figure 1. Precedence graph for the set J_5.

oriented diagonal and the i^{th} right oriented diagonal consist of the tasks J_{mj} $(0 \leq m \leq j)$ and J_{im} $(i \leq m \leq n)$, respectively. We say that the i^{th} right oriented and the j^{th} left oriented diagonals cross at task J_{ij}.

3. Systolic arrays. In a systolic array of processors the connectivity pattern between processors is fixed. Processors may have a local memory, but there is no shared memory. The processors run in lockstep and it is desirable that many processors are identical (i.e., execute (a copy) of the same program code). For this reason systolic arrays tend to be well suited for implementation on silicon chips. Now we will design a systolic array for the dynamic programming problem (1). We will assume that when processor P_i sends a value to P_j at time t, P_j must read this value at time t, otherwise it would be lost.

Our systolic array consists of $n(n+1)/2$ processors P_{ij} $(0 \leq i \leq j \leq n)$ that are arranged in a pyramid like the tasks J_{ij}. Processor P_{ij} performs task J_{ij}. We assume that processor P_{ij} contains the weight $w(i,j)$ in its register w.

Remark. In case of example 1 these values $w(i,j)$ can be computed easily. Assume that P_{ii} $(0 \leq i \leq n)$ receives from input the probabilities p_i and q_i. Obviously $w(i,j)=w(i,j-1)+p_j+q_j$. Thus, P_{ij} needs the weight of its neighbor on the right oriented diagonal and the value p_j+q_j from the left oriented diagonal. Observe that this latter value is independent of the index i. Thus, on the j^{th} left oriented diagonal the value p_j+q_j is sent forward, while on the right oriented diagonal each processor sends forward its own weight (see figure 2).

The calculation of the cost $c(i,j)$ must be dealt with more carefully. Processor P_{ij} needs optimal costs from processors on both diagonals that cross at it. P_{ij} must send forward the costs on the right and left oriented diagonals, but the cost $c(i,j)$ computed at P_{ij} must be sent in both directions (see figure 3). The obvious order to send the costs forward as they are received, appended by the newly computed $c(i,j)$, however, leads to an $\Omega(n^2)$ time systolic array: the order in which costs are received does not fit the order of computation of a new cost. Moreover, in this case processor $P_{i,i+k}$ needs $\Omega(k)$ memory.

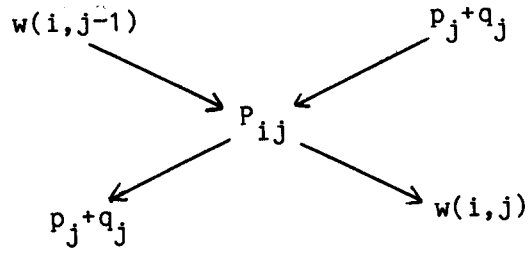

Figure 2. Communication pattern for the computation of weights.

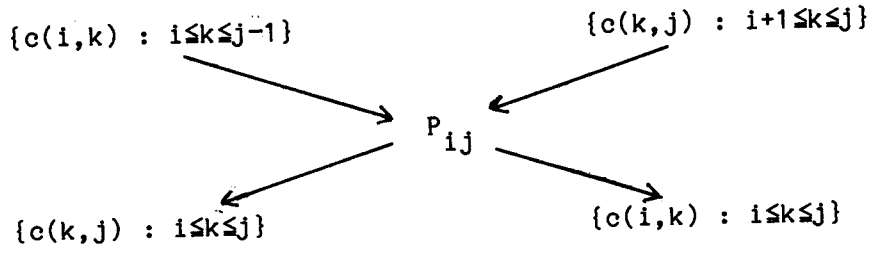

Figure 3. Skeleton of communication pattern to compute costs.

The problem is to find for each i,j a permutation π_{ij} of the numbers 1..j-i (i.e., the order in which P_{ij} receives the costs) such that

(1) $c(i,i+\pi_{ij}(k)-1)$ and $c(i+\pi_{ij}(k),j)$ are the k^{th} number received from the right and left oriented diagonals, respectively;

(2) knowing permutation π_{ij}, the permutations $\pi_{i,j+1}$ and $\pi_{i-1,j}$ can be determined on-line using a constant amount of memory, i.e. processor P_{ij} can determine the order in which costs are sent forward easily from the order in which costs are received.

Such permutations exist but we have to distinguish between processors of even and odd stages. Let $\pi_{i,i+2k}$ and $\pi_{i,i+2k+1}$ defined as

$$\pi_{i,i+2k}(2m+1)=k-m \quad 0\le m\le k \qquad \pi_{i,i+2k+1}(2m+1)=k+1+m \quad 0\le m\le k$$

$$\pi_{i,i+2k}(2m)=k+m \quad 1\le m\le k \qquad \pi_{i,i+2k+1}(2m)=k+1-m \quad 1\le m\le k$$

Figure 4 shows in what order $P_{2,6}$ receives and sends forward cost values. Remind that the sending orders established by $P_{2,6}$ is the receiving order of $P_{1,6}$ and $P_{2,7}$.

Processor P_{ij} can easily modify the order in which costs are sent forward. $P_{i,i+2k}$ interchanges on the left oriented diagonal the 2nd and 3rd, the 4th and 5th, etc. values; c(i,i+2k) and c(i+2k,i+2k) are sent last. As for the right oriented diagonal $P_{i,i+2k}$ interchanges the 1st and 2nd, the 3rd and 4th, etc. values and finally sends forward c(i,i+2k). Similarly, $P_{i,i+2k+1}$ interchanges on the left oriented diagonal the 1st and 2nd, the 3rd and 4th, etc. values and finally sends c(i,i+2k+1) and c(i+2k+1,i+2k+1). On the right oriented diagonal the 2nd and 3rd, the 4th and 5th, etc. values are interchanged and c(i,i+2k+1) is sent last.

Obviously a processor can perform this receive/send pattern on-line while using a constant amount of memory.

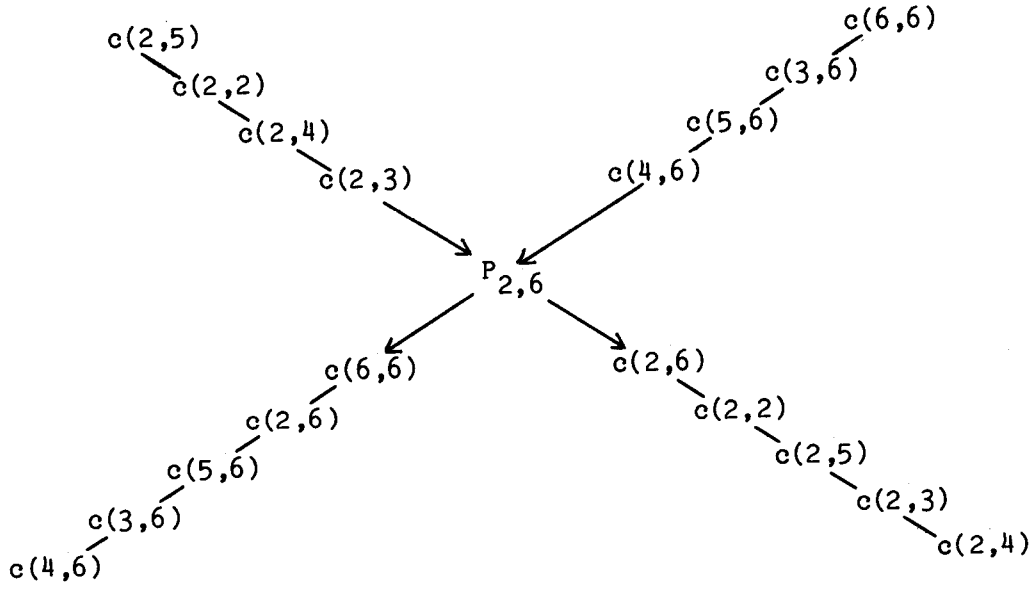

Figure 4. Communication pattern for the computation of costs.

Now the question arises whether we can develop programs for all processors such that this receive/send pattern is obtained, the cost at this processor can be computed and moreover that a value sent to a processor must be read by this processor at the same time. We may expect to find different programs for processors on odd and even stages. Each value received by a processor will be used three times by it: it must be read, it must be used for an inspection, and it must be sent forward. In one unit of time two values from different diagonals can be used for an inspection. Thus, to do for two values from both diagonals all the work at one processor requires at least 10 units of time. In order to prevent loss of efficiency no values should be swapped from one register to another. Algorithm C shows programs for processors at odd and even stages that will do the job. It is assumed that all commands on one line can be done in one unit of time. Less than 7 memory locations are needed when different programs are written for processors at stages $4k$, $4k+1$, $4k+2$ and $4k+3$ and the loops are untangled. The programs given in algorithm C satisfy all the requirements provided that each processor starts 6 units of time after the processors at the previous stage started (except for processors at stage 1, that must start 1 unit of time later than processors at stage 0 do). All this leads to it that processor P_{0n} must wait at most $6n-5$ units of time before it can start execution and it needs $5n+9$ units of time itself.

THEOREM 2. There is a systolic array for the dynamic programming problem (1) that runs in $O(n)$ time, using $n(n+1)/2$ processors with a constant amount of memory for each processor.

COROLLARY 3. There is a systolic array for the dynamic programming problem (1) using $p=n(n+1)/2$ processors and has a speed-up of $S_p=\Omega(p)$ and efficiency $E_p=\Omega(1)$.

However, when the dynamic programming problem satisfies the quadrangle inequality, these numbers are $S_p=\Omega(n)=\Omega(\sqrt{p})$ and $E_p=\Omega(1/\sqrt{p})$, which is not very satisfactory. Observe that the systolic array designed does not adhere to the strategies mentioned in the introduction: computa-

$P_{i,i+2k}$ $(k \geq 1)$

```
read(ROD,R[0];
C:=∞, F:=1;
read(LOD,L[F]);
wait;
C := min(C,R[1-F]+L[F]);
read(LOD,L[1-F]);
send(LOD,L[F]);
read(ROD,R[F]);
send(ROD,R[F]);
C := min(C,R[F]+L[1-F]);
to  k-1  do begin
    read(ROD,R[F]);
    send(ROD,R[1-F]);
    read(LOD,L[F]);
    send(LOD,L[F]);
    C := min(C,R[F]+L[F]);
    read(LOD,L[F]);
    send(LOD,L[1-F]);
    read(ROD,R[1-F]);
    send(ROD,R[1-F]);
    C := min(C,R[1-F]+L[F]); F:=1-F;
enddo;
C := C+W;
send(ROD,R[1-F]);
wait;
send(LOD,C);
wait;
wait;
send(LOD,L[1-F]);
wait;
send(ROD,C);
```

$P_{i,i+2k+1}$ $(k \geq 0)$

```
read(LOD,L[0];
C:=∞, F:=1;
read(ROD,R[F]);
wait;
C := min(C,L[1-F]+R[F]);
to  k  do begin
    read(ROD,R[1-F]);
    send(ROD,R[F]);
    read(LOD,L[F]);
    send(LOD,L[F]);
    C := min(C,R[1-F]+L[F]);
    read(LOD,L[F]);
    send(LOD,L[1-F]);
    read(ROD,R[F]);
    send(ROD,R[F]);
    C := min(C,R[F]+L[F]); F:=1-F
    enddo;
C := C+W;
send(ROD,R[F]);
wait;
send(LOD,C);
wait;
wait;
send(LOD,L[1-F]);
wait;
send(ROD,C);
```

P_{ii}

```
C:=0;
send(LOD,C);
wait;
send(ROD,C);
```

Algorithm C: programs for the systolic array for dynamic programming problem (1). LOD and ROD stand for left and right oriented diagonal, respectively.

tions for different stages are done simultaneously.

4. Using less than n^2 processors. In this section we will consider the case that $p<n^2$ processors are available. Obviously a linear time algorithm is then impossible. But we will design an algorithm that requires $O(n^3/p)$ parallel time, which is the best possible. We will also develop some ideas important for the next section. As a machine model, we will use the PRAM model: processors have a shared memory, processors may execute different programs and moreover processors can be synchronized, i.e., points in the program can be specified where the processor may only proceed when the other processors have arrived at the corresponding synchronization points in their programs.
A first approach would be to divide the tasks of J_n among the processors and to establish synchronization between each two consecutive stages. However this may lead to an $\Omega(n^2)$ algorithm.

THEOREM 4. Let A be a scheduling of the task system J_n with precedence relation $<$ on a set of processors such that
(i) each J_{ij} is assigned to exactly one processor, and
(ii) the execution of a task in stage k will only start when the execution of all tasks in stage k-1 are finished.
Then the execution of A requires at least $\Omega(n^2)$ time.

Proof. We will construct an example of an optimal binary search tree problem. Suppose n keys $a_1<\ldots<a_n$ are given. We will assign probabilities to these keys and the intervals between two consecutive keys such that there is a task at each stage i (i even) that requires to inspect at least 2i keys for being the root of the corresponding optimal subtree. Thus between the starts of stages i and i+1 (i even) there is a time difference of at least 2i. With at least n/2 even stages, total time will be $\Omega(n^2)$.

Let n be even. Let the search probabilities for the intervals between keys all be zero. Assign positive weights to $a_{n/2}$ and $a_{n/2+1}$ such that $a_{n/2+1}$ is the root of the optimal subtree $T_{n/2-1,n/2+1}$. Now suppose that weights are assigned to $a_{n/2-i+1}, \ldots, a_{n/2+i}$. Then weights can be assigned to $a_{n/2-i}$ and $a_{n/2+i+1}$ such that the optimal subtrees $T_{n/2-i-1,n/2+i}$, $T_{n/2-i,n/2+i+1}$ and $T_{n/2-i-1,n/2+i+1}$ are as

shown in figure 5. Then we have

$$R(n/2-i,n/2+i) = n/2+i, \quad R(n/2-i-1,n/2+i) = n/2-i \quad \text{and}$$

$$R(n/2-i,n/2+i+1) = n/2+i+1.$$

and as a consequence task $J_{n/2-i,n/2-i+2i}$ requires at least 2i keys that must be inspected to be the root. Thus for each i ($1 \le i \le n/2$) there is a task at stage 2i whose execution requires at least 2i control values for inspection. Summing over all even stages leads to $\Omega(n^2)$ control values that are inspected sequentially. Because each inspection requires a constant amount of time, the overall parallel time is $\Omega(n^2)$. Q.E.D.

In order to obtain an upperbound $O(n^3/p)$ even with $p=O(n^{1+\varepsilon})$ ($\varepsilon>0$) we must divide inspections of one task to different processors. The main idea is to number all inspections of each stage k from $I_1^{(k)}$, ..., $I_{(n-k+1)n}^{(k)}$. Then inspection $I_x^{(k)}$ is used for task $J_{i,i+k}$ with $i=\lceil x/n \rceil$ and the involved control variable uvar is uvar=mod(x,n). Thus, given the index of an inspection, we can easily determine the task it belongs to and the control variable involved. The total number of inspections in stage k is (n-k+1)n ($0 \le k \le n$).

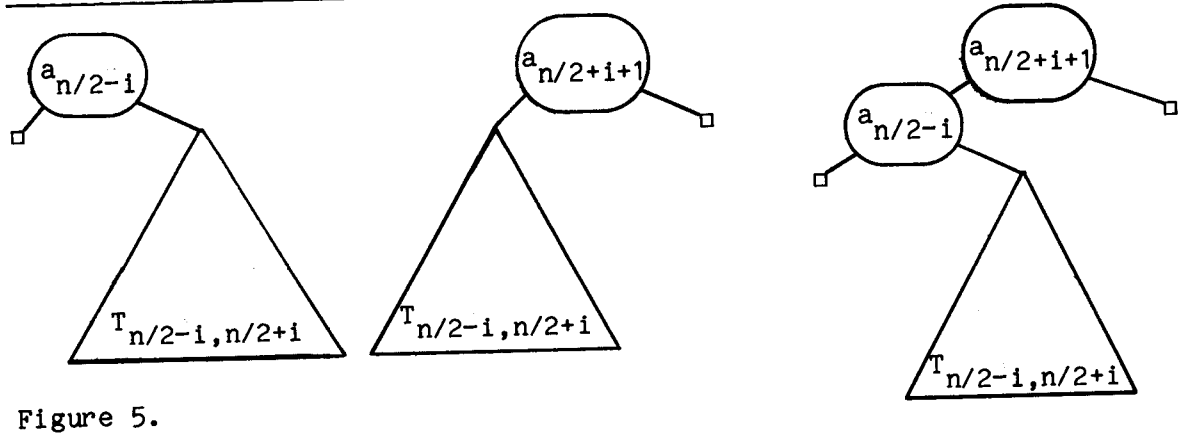

Figure 5.

When at stage k $p>(n-k+1)*n$, there are more processors than inspections. All inspections can be done in parallel, hence the time needed for it is bounded by a constant. The results of the inspections must be used to find the n-k+1 optimal costs. With more processors than results of inspections, the optimal costs can be found in $O(\log n)$ time.

When $n\leq p<(n-k+1)n$ each processor can perform $\lceil (n-k+1)n/p \rceil$ inspections and it performs inspections of at most 2 different tasks. Each processor can already take the minimum of all inspections it performs and that belong to the same task (we will call such a minimum a partial minimum).

Computation of the partial minima requires at most $O((n-k+1)n/p)$ parallel time. Suppose $P_{j_1}, \ldots, P_{j_m}$ compute partial minima for some task J. Then P_j $(j_1<j<j_m)$ computes only one partial minimum. We can use processors $P_{j_1}, \ldots, P_{j_m-1}$ for the computation of the optimal cost for task J.

In this way no processor is involved in the computation of more than one optimal cost and when m partial minima are given for one task, we can use m-1 processors to determine the optimal cost. Obviously this can be done in $O(\log p/(n-k+1))$ time.

Thus at stage k $(n\leq p<(n-k+1)n)$ the amount of time is $O(\frac{(n-k+1)n}{p} + \log_{\frac{p}{n-k+1}})$.

THEOREM 5. With $p=\Theta(n^{1+\varepsilon})$ $(0<\varepsilon<1)$, the dynamic programming problem (1) can be computed in time $O(n^3/p)$.

Proof. At stage k $O(\log n)$ time is needed when $p>(n-k+1)n$ and $O(\frac{(n-k+1)n}{p} + \log_{\frac{p}{n-k+1}})$ time otherwise. Taking the sum over all stages k $(1\leq k\leq n)$ yields the time bound. Q.E.D.

When n or less processors are available, the easiest way is to assign each task to one processor. Then at stage k each processor has to perform at most $\lceil (n-k+1)/p \rceil$ tasks, each requiring $O(n)$ time.

THEOREM 6. With $p \leq n$ the dynamic programming problem (1) can be solved in $O(n^3/p)$ time on p processors.

Proof. In stage k at most $O(n^2/p)$ time is needed. Summing over all stages yields the bound. Q.E.D.

Observe that in the latter two theorems most time is used for performing the inspections. Computation of the optimal costs out of the partial minima is only a minor term.

5. An $O(n^2/p + n \log \log p)$ algorithm. Though theorem 4 was proven for dynamic programming problems in general, it even holds in those cases that the quadrangle inequality is satisfied. In order to achieve an upper bound lower than $O(n^2)$ we must allow for assignments of parts of tasks J to different processors. When the quadrangle inequality holds, there are at most a linear (in n) number of inspections (cf. [5]). With $p=\Omega(n)$ we could assign inspections to processors in such a way that the number of inspections performed by one processor, is bounded by a constant. Unfortunately, the inspections can be divided very unevenly among the tasks of one stage (e.g. in the example used in the proof of theorem 4). As a consequence we must take more care of the computation of the optimal costs out of the partial minima.

Suppose that at stage k-1 the control variables at which the optimal costs are achieved, are stored in the array $R^{(k-1)}[0..n-k+1]$. Task $J_{q,q+k}$ consists only of inspections associated with control variable $R^{(k-1)}[q], \ldots, R^{(k-1)}[q+1]$. Let us therefore number the control variables as follows:

inspection		task	control variable
$I_i^{(k)}$	$1 \leq i \leq R^{(k-1)}[1]$	$J_{0,k}$	i
$I_i^{(k)}$	$R^{(k-1)}[1]+1 \leq i \leq R^{(k-1)}[2]+1$	$J_{1,k+1}$	i-1
$I_i^{(k)}$	$R^{(k-1)}[2]+2 \leq i \leq R^{(k-1)}[3]+2$	$J_{2,k+2}$	i-2

etc.

Thus we have:

Given i ($1 \le i \le 2n-k$); let q ($0 \le q \le n-k$) satisfy that

$$R^{(k-1)}[q]+q \le i \le R^{(k-1)}[q+1]+q \qquad (3)$$

Then inspection $I_i^{(k)}$ is used in task $J_{q,k+q}$ and involves control variable u_{i-q} (i.e., in example 1 key a_{i-q} is inspected for being the root of the optimal subtree $T_{q,k+q}$).

A search on $R^{(k-1)}$ seems to be involved when for an arbitrary i the q satisfying (3) must be found. However, things are easier when several consecutive values for i or consecutive stages k are involved.

LEMMA 7. Let q_i satisfy (3) for a given value i. Then $q_i \le q_{i+1} \le q_i+1$ and $q_i-1 \le q_{i-1} \le q_i$.
Proof. Obvious.

LEMMA 8. Let inspection $I_i^{(k)}$ be used for task $J_{q,k+q}$. Then inspection $I_i^{(k+1)}$ is used for either task $J_{q-1,k+q}$ or $J_{q,k+q+1}$.
Proof. $I_i^{(k)}$ is used for $J_{q,k+q}$. Hence $R^{(k-1)}[q]+q \le i \le R^{(k-1)}[q+1]+q$. With $R^{(k)}[q-1] \le R^{(k-1)}[q]$ and $R^{(k-1)}[q+1] \le R^{(k)}[q+1]$, this leads to $R^{(k)}[q-1]+q \le i \le R^{(k)}[q+1]+q$ and thus

$$R^{(k)}[q-1]+q-1 \le i \le R^{(k)}[q+1]+q$$

This is identical to i satisfies either $R^{(k)}[q-1]+q \le i \le R^{(k)}[q]+q-1$ or $R^{(k)}[q]+q \le i \le R^{(k)}[q+1]+q$. This latter statement is nothing else than that $I_i^{(k+1)}$ is used for either $J_{q-1,k+1+q-1}$ or $J_{q,k+1+q}$. Q.E.D.

These two lemmas make the following assignment of inspections of stage k to processors obvious.

P_j ($\mathrm{mod}(2n-k,p) \le j \le p$) will perform the inspections $I_i^{(k)}$ for all i with $\lfloor \frac{2n-k}{p} \rfloor * (j-1) + \mathrm{mod}(2n-k,p)+1 \le i \le \lfloor \frac{2n-k}{p} \rfloor * j + \mathrm{mod}(2n-k,p)$.

P_j ($1 \le j \le \mathrm{mod}(2n-k,p)$) will perform the inspections $I_i^{(k)}$ for all i with $\lceil \frac{2n-k}{p} \rceil * (j-1) + 1 \le i \le \lceil \frac{2n-k}{p} \rceil * j$.

LEMMA 9. Given the above assignment of inspections to processors. Given k, p, n and an arbitrary index x. Then we can determine in constant time which processor P_j performs inspection $I_x^{(k)}$.

Proof. The following formula does the job.

$$\begin{aligned} &\text{if} \quad x > \lceil (2n-k)/p \rceil \times \mathrm{mod}(2n-k,p) \\ &\text{then } j := \left\lceil (x - \mathrm{mod}(2n-k,p)) / \left\lfloor \frac{2n-k}{p} \right\rfloor \right\rceil \\ &\text{else} \quad j := \left\lfloor x / \left\lceil \frac{2n-k}{p} \right\rceil \right\rfloor \end{aligned}$$

Q.E.D.

With this assignment we have that a processor that performs the inspections $I_s^{(k)}, \ldots, I_t^{(k)}$ performs also the inspections $I_s^{(k+1)}, \ldots, I_t^{(k+1)}$ or $I_{s-1}^{(k+1)}, \ldots, I_{t-1}^{(k+1)}$, but no other inspections of stage k and k+1.

Now we have assigned the inspections of stage k very evenly to the processors. Before we present the precise data structure and algorithm, we have to solve another problem. Inspections of one task (state) can be assigned to different (consecutive) processors. These processors compute only partial minima for this task. Thus, the minimum (i.e., the optimal cost) of these partial minima must be computed. It even may happen that $\Omega(p)$ partial minima for one task are computed; (see the example in the proof of theorem 4). [7] and [6] presented parallel algorithms that compute the minimum of N numbers in $O(N/p + \log\log p)$ time if $1<p\leq N$ processors were available. We will use the algorithm of [6].

In the algorithm we distinguish in each stage k the following steps.

(1) Each processor must determine the indices of the inspections that it must perform (say $I_s^{(k)}, \ldots, I_t^{(k)}$).

(2) Each processor must determine the index qs such that inspection $I_s^{(k)}$ belongs to task $J_{qs,qs+k}$.

(3) Each processor P_j must determine the smallest index firsts of all processors that perform inspections for task $J_{qs,qs+k}$.

(4) Each processor must perform its inspections $I_s^{(k)}$, ..., $I_t^{(k)}$ and compute as many partial minima as there are tasks in stage k of which inspections are performed by this processor.

(5) Each processor P_j must determine the index qt such that inspection $I_t^{(k)}$ belongs to task $J_{qt,qt+k}$; each processor P_j must determine the largest index lastt of all processors that perform inspections for task $J_{qt,qt+k}$.

(6) Find for each task in stage k its optimal cost.

THEOREM 10. With $p \leq n$ the algorithm runs in $O(n^2/p + n \log\log p)$ parallel time.

Proof. The initialization of all relevant data and stage 1 can be done in $O(n^2/p)$ time. Then for each stage k ($2 \leq k \leq n$) we have:

Step (1) requires constant time.

Step (2) can be done in constant time (see lemma 8).

In step (3) P_j computes which processor performs inspection $I_x^{(k)}$ with $x = R^{(k)}(qs)+qs$. According to lemma 9 this can be done in constant time.

Each inspection in step (4) can be performed in constant time. With $O(n/p)$ inspections per processor per stage, step (4) needs at most $O(n/p)$ time.

Step (5) can be done in constant time (similar to step (2) and (3)).

With step (6) we must be more careful. Observe that each processor computes at most 2 partial minima (for tasks $J_{qs,qs+k}$ and/or $J_{qt,qt+k}$) and at most t-s+1 optimal costs. These optimal costs can be stored in the appropriate storage locations in $O(n/p)$ time.

Now suppose that the partial minima for a task J have been computed by P_{j_1}, ..., P_{j_m}. P_{j_1} and P_{j_m} might have computed partial minima for other tasks too, but the processors in between have not. Then processors P_{j_1}, ..., P_{j_m-1} will be involved in the computation of the optimal cost for task J. With this assignment no processor will be involved in the computation of more than one optimal cost. Thus, when y partial minima must be composed to one optimal cost, there are y-1 processors

to do this job. According to [6] it can be done in $O(\log\log(y-1)) \leq O(\log\log p)$ time.
Step (6) requires at most $O(n/p + \log\log p)$ time in each stage.

With n stages, the overall parallel time of the algorithm amounts to $O(n^2/p + n \log\log p)$ time. Q.E.D.

COROLLARY 11. With n processors the dynamic programming problem (1) satisfying the quadrangle inequality can be solved in $O(n \log\log n)$ time, thus achieving a speed-up of $S_n = \Omega(n / \log\log n)$ and an efficiency of $E_n = \Omega(1 / \log\log n)$.

The algorithm uses the following data structures in shared memory:

W a matrix of size [0..n]*[0..n] where W[i,j] ($i \leq j$) contains the weight w(i,j);

C a matrix of size [0..n]*[0..n]. C will eventually contain the optimal costs c(i,j) ($0 \leq i \leq j \leq n$).

partm a matrix of size [0..p]*[0..n]. During stage k partm[j,q] will contain the partial minimum computed by P_j for task $J_{q,q+k}$.

R an array of size [0..n]. During stage k R[i] contains the control variable for which the optimal solution for $J_{i,i+k-1}$ is achieved ($0 \leq i \leq n-k+1$).

partr a matrix of size [0..p]*[0..n]. partr[j,q] will contain during stage k the control variable for which the partial minimum partm[j,q] is achieved.

n the size of the dynamic programming problem.

p the number of processors.

All other data structures will be local to each processor. We assume that each processor P_j knows its index j.

```
Initialization and stage 1:
   for m:=⌊(n+1)/p⌋*(j-1)  to  minim ( ⌊(n+1)/p⌋*j -1, n)
     do  C[m,m] := 0  od;
   for m := ⌊n/p⌋*(j-1)  to  minim ( ⌊n/p⌋*j -1, n-1 )
     do  C[m,m+1] := W[m,m+1];  R[m] := m+1  od;
   remaind := mod(2n-2,p)
   size := ⌊(2n-2)/p⌋;    size1 := ⌈(2n-2)/p⌉;
   if  j > remaind
   then  s := size*(j-1) + remaind +1
   else  s := size1*(j-1) +1;
   qs := ⌊(s-1)/2⌋;

 stage 2..n:
    for  k := 2 to n
    do remaind := mod(2n-k,p);
       size := ⌊(2n-k)/p⌋;    size := ⌈(2n-k)/p⌉;
       if  j > remaind
       then  s := size*(j-1) + remaind +1;  t := size*j + remaind
       else  s := size1*(j-1) +1;    t := size1*j
       fi;
       if R[qs]+qs+1>s
       then   qs := qs-1;    if R[qs]+qs+1 > s  then qs := qs-1  fi
       fi;
       q := qs;    min := ∞;
       if  R[q]+q+q > size1*remaind
       then firsts := ⌈(R[q]+q+1-remaind)/size⌉
       else  firsts := ⌈(R[q]+q+1)/size1⌉
       fi;
       for r := s  to  t
       do  uvar := r-q-1;
           if min > C[q,uvar-1] + C[uvar,q+k]
           then  min := C[q,uvar-1] + C[uvar,q+k];
                 partr[j,q] := uvar-1
           fi;
           if r = R[q+1]+q+1
           then partm[j,q] := min;  min := ∞;  q := q+1
           fi
        od;
        if  R[q+1]+q+1 > remaind*size1
        then lastt := ⌈(R[q+1]+q+1-remaind)/size⌉
        else  lastt := ⌈(R[q+1]+q+1)/size1⌉
        fi;
        qt := q;
        (* now all partial minima has been computed *)
        if firsts = j
        then C[qs,qs+k] := partm[1,qs] + W[qs,qs+k];  R[qs] := partr[j,qs]
        fi;
        if lastt = j
        then C[qt,qt+k] := partm[1,qt] + W[qt,qt+k];  R[qt] := partr[j,qt]
        fi;
```

```
    for  q := qs+1  to  qt-1
    do  C[q,q+k] := partm[1,q] + W[q,q+k];  R[q] := partr[j,q]  od;
    if  lastt>j
    then P_j is involved in the computation of the minimum of
         partm[st..lastt,qt] is which st=1 if qs≠qt and st=firsts
         otherwise. Thus all information needed for the application
         of the algorithm of [6] is available to P_j.
od;
```

Algorithm D: Parallel algorithm to solve a dynamic programming problem with the quadrangle inequality.

REFERENCES.

[1] Al-Dabass, D., Two methods for the solution of the dynamic programming algorithm on a multiprocessor cluster, Opt. Contr. Applic. & Methods 1 (1980), pp. 227-238.

[2] Aho, A.V., J. Hopcroft and J. Ullmann, The design and analysis of computer algorithms, Addison-Wesley, Reading Mass., 1974.

[3] Bertsekas, D.P., Distributed dynamic programming, IEEE Trans. Autom. Contr., AC-27 (1982), pp. 610-616.

[4] Cati, J., M. Richardson and R. Larson, Dynamic programming and parallel computers, J. Opt. Theor. Applic. 12 (1973), pp. 423-438.

[5] Knuth, D.E., Optimum binary search trees, Acta Inform. 1 (1971), pp. 14-25.

[6] Shiloach, Y. and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, J. Algor. 2 (1981), pp. 88-102.

[7] Valiant, L.G., Parallelism in comparison problems, SIAM J. Comput. 4 (1975), pp. 348-355.

[8] Yao, F.F., Efficient dynamic programming using quadrangle inequalities, Proc. 12th Ann. ACM Symp. Theory of Comp., ACM, 1980, pp. 429-435.