On a Sublinear Time Parallel Construction of Optimal Binary Search Trees

Marek Karpinski[1]* and Wojciech Rytter[2]**

[1] Department of Computer Science, University of Bonn
53117 Bonn, Germany
and
International Computer Science Institute, Berkeley
CA, USA
[2] Department of Computer Science, University of Warsaw
02–097 Warsaw, Poland

Abstract. We design an efficient sublinear time parallel construction of optimal binary search trees. The efficiency of the parallel algorithm corresponds to its total work (the product $time \times processors$). Our algorithm works in $O(n^{1-\epsilon}\log n)$ time with the total work $O(n^{2+2\epsilon})$, for an arbitrarily small constant $0 < \epsilon \leq \frac{1}{2}$. This is optimal within a factor $n^{2\epsilon}$ with respect to the best known sequential algorithm given by Knuth, which needs only $O(n^2)$ time due to a *monotonicity* property of optimal binary search trees, see [6]). It is unknown how to explore this property in an efficient NC construction of binary search trees. Here we show that it can be effectively used in sublinear time parallel computation. Our improvement also relies on the use (in independently processed small subcomputations) of the parallelism present in Knuth's algorithm. The best known sublinear time algorithms for the construction of binary search trees (as an instance of a more general problem) have $O(n^3)$ work for time larger than $n^{3/4}$, see [3] and [7]. For time $\sqrt{n}$ these algorithms need n^4 work, while our algorithm needs for this time only n^3 work, thus improving the known algorithms by a linear factor. Also if time is $O(n^{1-\epsilon})$ and ϵ is very small our improvement is close to $O(n)$. Such improvement is similar to the one implied by the *monotonicity* property in sequential computations (from n^3 sequential time for a more general *dynamic programming* problem to n^2 time for the special case of optimal binary search trees).

1 Introduction

One of the main advantages of parallelism is the time reduction, however this is usually done at the expense of the *total work* (the product $time \times processors$). For many algorithmic problems the following fact can be observed when we solve

* Research supported in part by the DFG Grant KA 673/4–1, ESPRIT BR Grants 7079 and ECUS030, and by the Volkswagen–Stiftung. Email: marek@cs.uni-bonn.de.
** Research partially supported by the Grant KBN 2–1190–91–01.

them in parallel: the slower is the computation the better is the total work of the algorithm. Usually from the point of view of the total work *only* the best ones are algorithms using only one processor (no parallelism). Also sublinear time algorithms have usually much better total work than NC algorithms. High speed involves a computation *in advance* of a lot of redundant objects. The size of the set of objects is an important factor. The dynamic programming gives a typical class of problems for which all three classes of computations (sequential, sublinear, NC) differ substantially in the algorithmic effciency (in the sense of total work).

The basic problem in this class is the construction of optimal binary search trees. We design an algorithm especially suited for this problem. Our sublinear time parallel algorithm is a combination of parallel and sequential processing. The whole computation is partitioned into a sublinear number of phases, processed successively in a sequential manner. Each of these phases is performed in a parallel way. The number of processors is substantially reduced since each phase is processing much smaller sets of objects. Two main features of our algorithm which improve its total work are:

1. splitting a crucial part (here preprocessing) into a number of independent sequential subcomputations, and apply to each of them the best known sequential algorithm (in our case the Knuth's algorithm);
2. using (in a parallel way) the basic property of a given problem explored in the best sequential algorithms, in our case a *monotonicity* property.

The construction of optimal binary search trees (the OBST problem) is an important algorithmic problem which has so far resisted any really efficient $\mathcal{NC}$ implementation, though Knuth gives a quadratic time sequential algorithm [6]. Only a special case (alphabetic trees) can be solved by a really efficient (in the sense of total work) NC algorithm, see [8]. The Knuth's algorithm uses a *monotonicity property* of optimal binary search trees (defined later). It is unknown how to use a similar property to reduce number of processors in polylogarithmic time computations. The best upper bound in polylogarithmic time computations is close to n^6, see [9], and is certainly too large. It applies to much larger class of problems which are instances of the dynamic recurrences problem. This generality is probably a cause of the inefficiency. Optimal binary search trees have special properties. The way to an improvement is a suitable use of these properties.

The binary search tree problem is a special case of the more general *dynamic programming* problem. In [3] and [7] (independently) it was shown that the dynamic programming recurrences can be solved in $n^{1-\epsilon}$ time with cubic total work for $\epsilon \geq 1/4$. (A slightly worse algorithm was presented in [4]).

So these algorithms worked in $n^{3/4}$ time with $O(n^3)$ work. However for smaller time the work increases, for example if time is $O(\sqrt{n})$ then the total work in these algorithms was of order n^4.

In our algorithm the work is reduced. For example if time is $O(n^{3/4})$ then the work done by our algorithms is $n^{2.5}$ and if time is $O(\sqrt{n})$ then this work is

n^3. In the latter case we have linear factor improvement. A similar improvement by a linear factor occurs in sequential computation of the considered problem (compared to the general *dynamic programming* problem).

Statement of the OBST problem. We use terminology from [5], pages 434-435. Let $\beta = (K_1, \ldots K_n)$ be a sequence of n weighted items (keys), which are to be placed in a binary search tree. We are given $2n+1$ weights (probabilities): $p_1, p_2, \ldots, p_n, q_0, q_1, \ldots, q_n$ where

- p_i is the probability that K_i is the search argument;
- q_i is the probability that the search argument lies between K_i and K_{i+1}.

We assume that K_i's are stored in internal nodes of the binary search tree and in external nodes special items are stored. The i-th special item K'_i corresponds to all keys which are strictly between K_i and K_{i+1}.

If T is a binary search tree with n internal nodes, where the i-th internal node (in in-order) is labeled K_i, and the external nodes correspond to sequence of special keys K'_i's, then define the *cost* of T as follows:

$$cost(T) = \sum_{i=1}^{n} p_i \cdot \ell(K_i) + \sum_{i=0}^{n} q_i \cdot \ell(K'_i). \tag{1}$$

where $\ell(K)$ is the *level* of K in T, defined to be the distance (number of internal nodes on the path) from the root. The OBST problem is then the problem of finding that tree T of minimum cost for a given sequence of items.

Our main result is:

Theorem 1. *An optimal binary search tree can be constructed in* $O(n^{1-\epsilon}/\log(n))$ *time with* $O(n^{2+2\epsilon})$ *total work, where* $\epsilon > 0$ *is an aritrarily small constant* $0 < \epsilon \leq \frac{1}{2}$.

Denote by $obst(i,j)$ an optimal binary tree whose keys correspond to the interval $int(i,j) = [K_{i+1} \ldots K_j]$ and denote by $cost(i,j)$ the cost of such tree. Let

$$w(i,j) = p_{i+1} + \ldots + p_j + q_i + \ldots + q_j.$$

The costs obey the following *dynamic programming recurrences* for $0 \leq i < j \leq n$:

$$cost(i,j) = w(i,j) + \min\{cost(i,k-1) + cost(k,j) \ : \ i < k \leq j \}. \tag{2}$$

Denote the smallest value of k which minimizes the above equation by $CUT(i, j)$ with boundary values $cost(i,i) + 0$. This is the first point giving an optimal decomposition of $obst(i,j)$ into two smaller (son) subtrees. Optimal binary search trees have the following crucial property (proved in [6]):
monotonicity property:

$$i \leq i' \leq j \leq j' \Longrightarrow CUT(i,j) \leq CUT(i',j').$$

Sequentially the values of $cost(i,j)$ are computed by tabulating them in an array. Such tabulation of costs of smaller subproblems is the basis of the *dynamic programming* technique. We use the same name *cost* for this array and call it the *dynamic programming table*. It can be computed in $O(n^3)$ time, by procesing diagonal after diagonal, starting with the central diagonal, see [1]. In case of optimal binary search trees this can be reduced to $O(n^2)$ using additional tabulated values of the cuts in table CUT (see [10]).

Once the table $cost(i,j)$ is computed then the construction of an optimal tree can be done very efficiently in parallel. The following (easy to see) result was also observed in [2]:

Lemma 2. *If the table of costs is computed then an optimal tree can be constructed in $O(\log n)$ time with $n^2/\log(n)$ processors.*

The structure of our algorithm is to mimic the sequential computation, however instead of computing one diagonal after the other we advance in larger steps. Let $\Delta = n^\epsilon$. Divide the upper (the only relevant) part of the dynamic programming table into n/Δ *strips* $S_1, \ldots, S_{n/\Delta}$, each one consisting of Δ consecutive diagonals, see Figure 1.

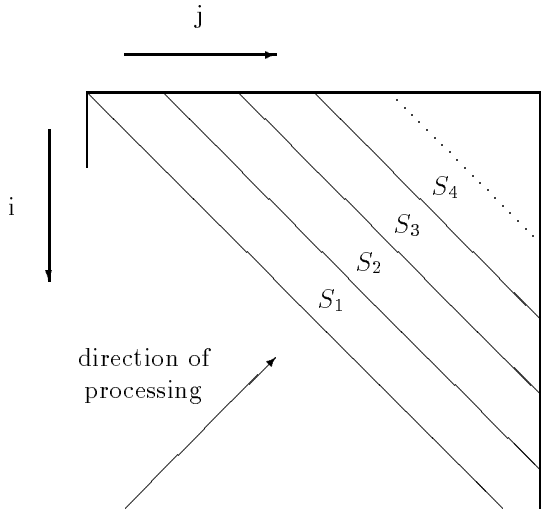

Fig. 1. The dynamic programming table and its partition into the strips.

In one phase we compute in logarithmic time the part of the table corresponding to the k-th strip S_k. In order to do it we need some preprocessing.

This involves so called *special subtrees*, defined later. The whole structure of the algorithms is:

Algorithm
1. Preprocessing:
   construct all optimal special subtrees;
2. **for** k = 1 **to** n/Δ **do** {in a sequential way}
   k-th phase: compute in parallel the strip S_k;

2 Preprocessing: Computing Costs of the Special Subtrees.

The crucial point in the preprocessing is to explore parallelism which is present in the Knuth's algorithm: elements on the same diagonal of the table *cost* can be computed independently (in parallel). The total work of the Knuth's algorithm is proportional to the number of computed objects.

We explain shortly how the time can be reduced to $n \cdot \log(n)$ without changing the total work. The diagonals of the *dynamic programming* table are computed one after the other exactly as they are computed in the Knuth's algorithm. In the computation of a given value by the formula (2) the minimization is restricted to the range of k's bounded from left and right by the cut values of the neighbouring entries (due to *monotonicity*). This guarantees that the total work for a fixed diagonal is linear (by the same argument as in [6]). All values on a fixed diagonal are computed simultaneously. Additionally the table of cuts is computed. We refer the reader to [6] for details. This implies the following fact.

Lemma 3. *The dynamic programming table for the OBST problem can be computed in $n \cdot \log n$ time with $n/\log(n)$ processors.*

Let $\beta = (K_1, \ldots K_n)$ be an initial sequence of n keys. For each $l < r$ denote by $\beta_{l,r}$ the sequence of keys $K_{l-\Delta}, \ldots, K_l, K_r, \ldots, K_{r+\Delta}$ with the weights for K_i's as before, however the weights of intervals (of special keys corresponding to intervals between consecutive *regular* keys) are $q_{l-\Delta-1}, \ldots, q_{l-1}, w(l,r), q_{r+1}, \ldots q_{r+\Delta}$. A *special binary subtree* with respect to (l,r) is any optimal binary subtree for any subinterval of $\beta_{l,r}$ of size at most $2 \cdot \Delta + 2$. An example of such tree is illustrated in Figure 2.

In other words special subtrees result from the subtrees for the original problem by cutting a large *gap*. The remaining part is bounded by 2Δ, the distance at most Δ to the left and to the right of the *gap*. Each *gap* is a large subtree removed from an original tree and replaced by the weight of removed items. An example of such subtree is also given in Figure 3 (the tree T3).

Denote by $cost_{l,r}(i,j)$ the cost of optimal special subtree with respect to (l,r), whose first key is K_{i+1} and the last is K_j.

There are $n(n-1)/2$ pairs l, r (potential *gaps*) and $\approx 2\Delta^2$ potential subintervals of each sequence $\beta_{l,r}$. We compute the cost of a single optimal subtree

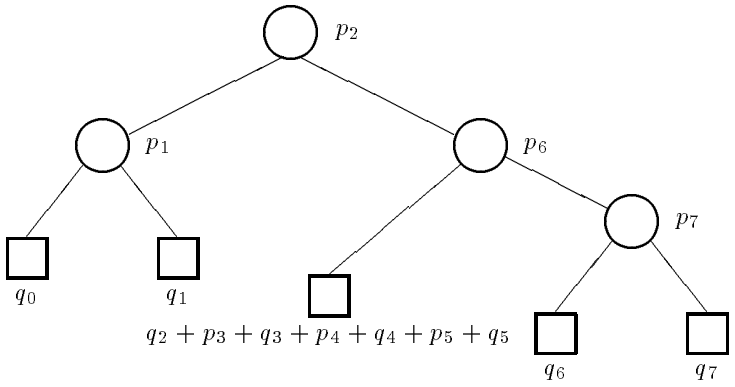

Fig. 2. An example of a special subtree with respect to $(l, r) =$ (2,6). The keys are in the circles and the intervals between the keys correspond to square nodes. The tree covers interval $(1, 7)$ with the *gap* between keys numbered $l = 2$, $r = 6$.

for each small subinterval of each sequence $\beta_{l,r}$. Altogether there are $n^{2+2\cdot\epsilon}$ such special subtrees. The preprocessing consists in computing costs of all optimal special subtrees. The best work (proportional to the total number of such trees) is achieved due to the Knuth's algorithm.

Lemma 4. *The values $cost_{l,r}(i, j)$ of the costs of all optimal special subtrees can be computed in $n^{\epsilon} \cdot \log(n)$ time using $n^{2+\epsilon}/\log(n)$ processors.*

Proof. We can compute costs of all optimal special subtrees for a fixed sequence $\beta_{l,r}$ in $n^{\epsilon}\log(n)$ time with $n^{\epsilon}/\log(n)$ proceessors by the algorithm from Lemma 2.1. We apply the algorithm from Lemma 2.1 to all special sequences $\beta_{l,r}$. For fixed (l, r) the size of the problem is $n' = O(\Delta)$ and the corresponding programming table contains $O(\Delta^2)$ entries. Lemma 2.1 gives a computation in time $n' \cdot \log(n')$ with $O(n'/\log(n'))$ processors for fixed (l, r). Summing over all possible pairs (l, r) (there is a quadrtaic number of such pairs) we have the required complexity. This completes the proof.

3 Implementation of a Single Phase of the Algorithm.

Let Γ_k be the set of all trees $obst(i, j)$ for $(i, j) \in S_k$. Assume we have computed the part of the dynamic programming table corresponding to strips $S_1, \ldots, S_k$. This means that we know the costs of all optimal trees in $\Gamma_{k'}$ for $k' \leq k$ and the CUT–values.

We describe how to compute the costs in the strip S_{k+1}. This is the $(k+1)$-st phase of the algorithm.

Each tree in Γ_{k+1} can be decomposed here in a way illustrated in Figure 3. $T1$, $T2$ are in $\Gamma_1 \cup \ldots \cup \Gamma_k$ and $T3$ is a special subtree resulting by replacing the interval (l,r) by a single external node of weight $w(l,r)$. It is easy to see that $T3$ is an optimal special subtree, its cost is already precomputed efficiently due to lemma 2.2. The costs of subtrees $T1$ and $T2$ are also already computed (the costs are in preceeding strips).

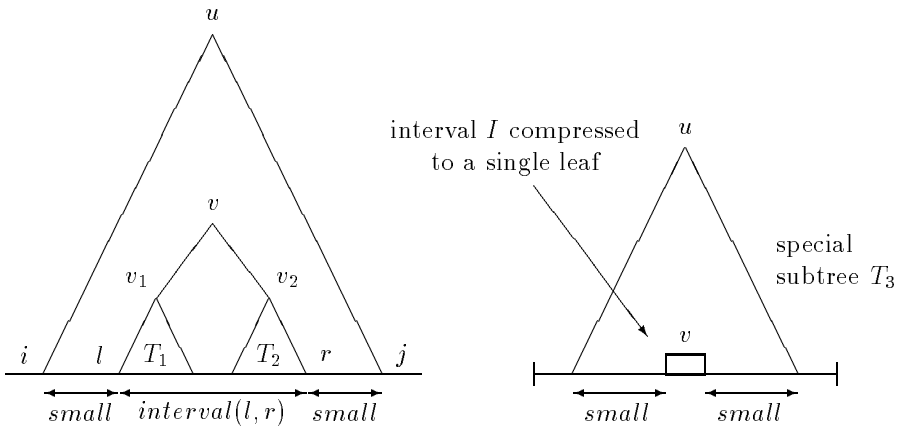

Fig. 3. A decomposition of the subtree in Γ_{k+1}. The subtrees $T1$, $T2$ rooted at $v1$, $v2$ are in $\Gamma_1 \cup \Gamma_2 \cup \ldots \Gamma_k$. A special tree $T3$ results by replacing the interval I by a single leaf whose weight equals the total weight $w(l,r)$ of I.

Let us fix k. For $(i,j) \in S_{k+1}$ define $cost'(i,j)$ as follows. If there is an optimal tree over the interval (i,j) such that both subtrees of its root v cover intervals in S_k then $cost'(i,j) = cost(i,j)$, the tree rooted at v in Figure 3 is of the above type. Otherwise let $cost'(i,j)$ be equal a cost of any binary search tree covering the interval (i,j).

Lemma 5. (Applying monotonicity property)
Assume the costs in all strips $S_1 \cup \ldots \cup S_k$ has been computed. Then the values $cost'(i,j)$ for all $(i,j) \in S_{k+1}$ can be computed in the $(k+1)$-st phase in $\log n$ time with $O(n^{1+2\cdot\epsilon})$ total.

Proof. Fix a p-th diagonal D of the strip S_{k+1}, see Figure 4. Divide D into Δ subsequences D_q, each one consists of points on D such that the distance of each consecutive points is Δ. We refer to Figure 4, where such subsequence consisting of points $x_1, x_2, \ldots$ is shown. Consider the subsequence $y_1, y_2, \ldots$ of the last diagonal of the preceeding strip. Assume we have computed the costs

for this points as well as their values $CUT(y_q)$'s. Then due to the monotonicity property we know that:

$$CUT(y_1) \leq CUT(x_1) \leq CUT(y_2) \leq CUT(x_2) \leq CUT(y_3) \leq CUT(x_3) \ldots \quad .$$

Hence to compute the values $cost'(x_1)$ we apply formula (2) and we have only to search in an interval of size $CUT(y_2) - CUT(y_1)$. Similarly the value of $cost'(x_2)$ is computed by minimizing over an interval of size $CUT(y_3) - CUT(y_2)$ etc. Altogether we need $O(n)$ processors for a single subsequence D_q. The minimization is done in logarithmic time. We have Δ subsequences D_q in a fixed p-th diagonal of next strip. There are Δ values of p. Altogether $n \cdot \Delta \cdot \Delta$ processors are enough, which is $O(n^{1+2\cdot\epsilon})$. This complete the proof.

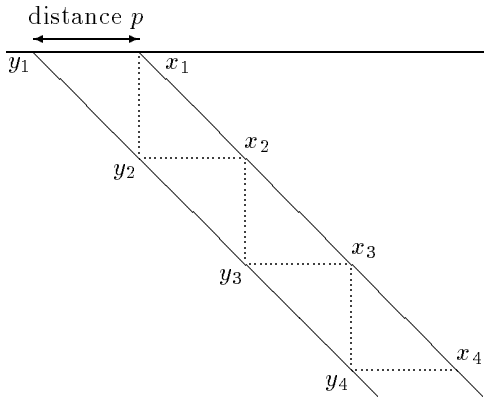

Fig. 4. Using information about cuts in Lemma 3.1.

The costs of points $(i,j) \in S_{k+1}$ can be computed by the formula:
(*) $\quad cost(i,j) = \min\{cost_{l,r}(i,j) + cost'(l,r) \ : \ (i,j) \supseteq (l,r) \in S_{k+1}\}$.
Then the successive phase is implemented as follows:

The (k+1)-th phase:
1. compute the values $cost'(i,j)$ for all $(i,j) \in S_{k+1}$;
2. compute in parallel the values $cost(i,j)$ for all $(i,j) \in S_{k+1}$ using the formula $(*)$.

Step 1 can be done in logarithmic time with $n^{1+2\cdot\epsilon}$ processors due to Lemma 3.1.

The total work done in Step 2 can be estimated as follows. There are $n^{1+\epsilon}$ points $(i,j) \in S_{k+1}$, for each of them we have to perform minimization (according to the formula (*)) over all (l,r) satisfying:

$$(i,j) \supseteq (l,r) \in S_{k+1} \ .$$

There are $n^{2\cdot\epsilon}$ such pairs (l,r) for a fixed (i,j). Altogether we need the same number of operations as the number of considered *4-tuples* (i,j,l,r). Hence $O(n^{1+3\cdot\epsilon})$ total work in a single phase is enough.

There are $n^{1-\epsilon}$ phases. Each of them needs $O(n^{1+3\cdot\epsilon})$ work. It gives together the total work of order $n^{2+2\cdot\epsilon}$ as claimed in Theorem 1.1. This completes the proof of our main result.

Conclusion.
We have presented a sublinear time algorithm which is quite close to an optimal one (with respect to the best sequential algorithm). A natural open problem is to find a (sublinear time) algorithm working in $n^{1-\epsilon}$ time, for some $\epsilon > 0$, whose total work is quadratic. Another important problem is the design of an efficient NC algorithm for the general OBST problem. Even an improvement $O(n^{6-\epsilon})$ in the number of processors would be of considerable interest (cf., also, [2]).

References

1. A.Aho, J.Hopcroft, J.Ullman, *Introduction to the design and analysis of computer algorithms* (1974), Addison–Wesley.
2. M. J. Atallah, S.R̃. Kosaraju, L. L. Larmore, G. L. Miller, and S–H. Teng, *Constructing trees in parallel*, Proceedings of the 1^{st} ACM Symposium on Parallel Algorithms and Architectures (1989), pp. 499–533.
3. Z. Galil, K. Park, *Parallel algorithms for dynamic programming recurrences with more than $\mathcal{O}(1)$ dependency*, Manuscript.
4. S. Huan, H. Liu, V. Viswanathan, *A sublinear time parallel algorithms for some dynamic programming problems*, Proceedings of the 1990 International Conference on Parallel Processing 3 (1990), pp. 261–264.
5. D. E. Knuth, *The Art of computer programming, Volume 3: Sorting and searching* (1973), Addison–Wesley.
6. D. E. Knuth, *Optimum binary search trees*, Acta Informatica 1 (1971), pp. 14–25.
7. L. Larmore, W. Rytter, *Efficient sublinear time parallel algorithms for the recognition for dynamic programming problems and context–free recognition*, in: STACS'92, Lecture Notes of Computer Science 577 (1992), Springer Verlag, pp.121–132
8. L. Larmore, T. Przytycka, W. Rytter, *Parallel construction of optimal alphabetic trees*, in: SPAA'93.
9. W. Rytter, *Efficient parallel computations for some dynamic programming problems*, Theoretical Computer Science 59 (1988), pp. 297–307.
10. F. F. Yao. *Efficient dynamic programming using quadrangle inequalities*, Proceedings of the 12^{th} ACM Symposium on Theory of Computing (1980), pp. 429–435.

This article was processed using the LaTeX macro package with LLNCS style