Deterministic Parallel Backtrack Search ⋆

Kieran T. Herley

Department of Computer Science, University College Cork, Cork, Ireland
k.herley@cs.ucc.ie

Andrea Pietracaprina, Geppino Pucci,

Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy
{andrea,geppo}@artemide.dei.unipd.it

Abstract

The *backtrack search problem* involves visiting all the nodes of an arbitrary binary tree given a pointer to its root, subject to the constraint that the children of a node are revealed only after their parent is visited. We present a fast, deterministic backtrack search algorithm for a p-processor COMMON CRCW-PRAM, which visits any n-node tree of height h in time $O\left((n/p+h)(\log\log\log p)^2\right)$. This upper bound compares favourably with a natural $\Omega(n/p+h)$ lower bound for this problem. Our approach embodies novel, efficient techniques for dynamically assigning tree-nodes to processors to ensure that the work is shared equitably among them.

Key words: Backtrack search. Load balancing. PRAM model. Parallel algorithms.

1 Introduction

Several algorithmic techniques, such as those employed for solving many optimization problems, are based on the systematic exploration of a tree, whose internal nodes correspond to partial solutions (growing progressively more refined with increasing depth) and whose leaves correspond to feasible solutions. In this paper, we are concerned with the implementation of tree explorations

⋆ This research was supported, in part, by the EC ESPRIT III Basic Research Project 9072-GEPPCOM; by the CNR of Italy under Grant CNR96.02538.CT07; and by MURST of Italy under Project MOSAICO. The results in this paper appeared in preliminary form in the *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming, Paderborn, Germany, July 1996.*

on shared-memory parallel machines. Specifically, we consider the *backtrack search problem*, which involves visiting all the nodes of a tree $\mathcal{T}$ subject to the constraints that (1) initially only the root of $\mathcal{T}$ is known to the processors, and (2) the children of a node are made known only after the node itself is visited. Moreover, the structure of $\mathcal{T}$, its size n and its height h are unknown to the processors.

We assume that a node can be visited (and its children revealed) in constant time. Since $\Omega(n)$ work is needed to visit n nodes and since any tree of height h contains a path of h nodes whose visit times must form a strictly increasing sequence, it follows that any algorithm for the backtrack search problem requires $\Omega(n/p + h)$ time on a p-processor machine.

A number of works on parallel backtrack search have appeared in the literature. Randomized algorithms have been developed for the completely-connected network of processors [KZ93,LAB93] and the butterfly network [Ran94], which run, optimally, in $O(n/p+h)$ steps, with high probability. It should be noted, however, that the butterfly algorithm focuses on the number of "node-visiting" steps and does not fully account for overhead due to manipulations of local data structures. A deterministic algorithm is given in [KP94], which runs in $O(\sqrt{ph})$ time on a $\sqrt{p} \times \sqrt{p}$ mesh, provided that $n = O(p)$. It is not clear whether this latter algorithm can be extended to work for larger tree sizes. The relationship between computation and communication for the exploration of trees arising from irregular divide-and-conquer computations has been studied in [WK91]. A number of related problems have also been addressed in the literature, such as branch-and-bound [Ran90,KZ93,LAB93,KP94,HPP99a,HPP99b] and dynamic tree embeddings [AL91,BGLL91,LNRS92].

In this paper, we present a deterministic PRAM algorithm for backtrack search whose running time is within a triply-logarithmic factor of the natural lower bound discussed above. Our main result is summarized in the following theorem.

Theorem 1 *There is a deterministic algorithm running on a p-processor COMMON CRCW-PRAM that performs backtrack search on any n-node bounded-degree tree of height h in $O\left((n/p + h)(\log\log\log p)^2\right)$ time, in the worst case.*

Ours is the first efficient, deterministic PRAM algorithm that places no restrictions on the structure, size or height of the (bounded-degree) tree to which it is applied, and whose running time faithfully accounts for all costs. The algorithm performs an optimal number of $O(n/p + h)$ parallel "node-visiting" steps, while the $O((\log\log\log p)^2)$ multiplicative factor in the running time captures the average overhead per step required to ensure that the workload is equitably distributed among the processors. As a consequence, our algorithm

would become optimal if the cost of a node visit were $\Omega((\log\log\log p)^2)$, which is likely to be the case in typical applications of backtrack search, where every node represents a complex subproblem to be solved.

The rest of the paper is organized as follows. Section 2 provides a number of basic definitions and discusses a simple, direct approach to backtrack search which our algorithm uses in combination with a more sophisticated strategy to attain efficiency. The high-level structure of our algorithm is described in Section 3, while Section 4 provides a detailed description of the key routine that performs node visits and load balancing. In Section 5 we argue the generality of our approach by discussing how it can be adapted to schedule straight-line computations represented by bounded-degree DAGs. Section 6 closes the paper with some final remarks.

2 Preliminaries

Our algorithm is designed for the COMMON CRCW PRAM model of computation, which consists of p processors and a shared memory of unbounded size. In a single step, each processor either performs a constant amount of local computation or accesses an arbitrary cell of the shared memory. In the COMMON CRCW variant of the PRAM, concurrent reads are permitted as are concurrent writes, provided that all competing processors write the same value [JáJ92].

Let $\mathcal{T}$ be the tree to be visited. For simplicity, we assume that the tree is binary, although our results can be immediately extended to the more general class of bounded-degree trees. For concreteness, we suppose that each node is represented in memory by means of a *descriptor*. Initially, only the descriptor of the root is available in the shared memory of the PRAM at a designated location. The descriptor of any other node is generated only by accessing the descriptor of the node's father. A *visit* to a node involves accessing its descriptor, and generating and storing the descriptors of its children (if any). As mentioned before, a node visit is assumed to take constant time.

A straightforward strategy to solve the backtrack search problem is to visit the tree in a breadth-first, level-by-level fashion. An algorithm based on such a strategy would proceed in phases, where each phase visits all the nodes at a certain level and evenly redistributes their children among the processors, to guarantee that the overall number of parallel visiting steps is at most $n/p + h$. (Here the term parallel visiting step refers to a k-tuple ($k \leq p$) of simultaneous visits to distinct tree nodes performed by distinct PRAM processors.) A perfectly balanced redistribution of tree nodes among processors between successive parallel visiting steps can be accomplished deterministically using

simple parallel prefix sums [JáJ92], yielding an $O(n/p+h\log p)$ overall running time for backtrack search. Note that this strategy also works for the weaker EREW PRAM variant, where concurrent read/write accesses are not allowed.

In fact, an asymptotically optimal number of $\Theta(n/p + h)$ parallel visiting steps can still be achieved without perfect balancing, by requiring that the nodes at any level of the tree be only "approximately redistributed" among the processors, that is, the nodes a processor is given must be at most a constant factor more than what it would receive with perfect balancing. An approximate redistribution can be attained by using the following result by Goldberg and Zwick.

Fact 2 ([GZ95]) *For an arbitrary sequence of p integer values $a_0, a_1, \ldots, a_{p-1}$, the* approximate prefix sums $b_0, b_1, \ldots, b_{p-1}$ *with $\sum_{j=0}^{i} a_j \leq b_i \leq (1+\epsilon)\sum_{j=0}^{i} a_j$, where $\epsilon = o(1)$, and $b_i \geq b_{i-1} + a_i$ can be determined in $O(\log\log p)$ worst-case time on a p-processor COMMON CRCW-PRAM.*

By employing the approximate prefix sums to implement node redistribution after visiting each level of the tree, we get a deterministic $O(n/p+h\log\log p)$-time algorithm for the backtrack search problem on a p-processor COMMON CRCW-PRAM, for any values of n, h and p.

In the next sections we devise a more sophisticated strategy which outperforms the above simple one for trees where $n = o(ph\log\log p/(\log\log\log p)^2)$. This asymptotic improvement results in near- optimal performance through careful "load-balancing" techniques without excessive global communication.

3 A High-Level View of the Algorithm

Our algorithm proceeds in a quasi-breadth-first fashion. Let the tree nodes be partitioned into h *levels*, where the nodes of one level are all at the same distance from the root. The exploration process is split into *stages*, each of which visits a *stratum* of the tree consisting of $\ell = \Theta(\log\log p)$ consecutive levels. At the beginning of a stage, all nodes at the top level of the stratum are (approximately) distributed among the processors. Note that the previously mentioned straightforward strategy based on approximate prefix sums visits any stratum of size $m = \Omega(p\ell^2)$ optimally. Therefore, we focus on techniques to cope efficiently with smaller strata.

Consider a stage visiting a stratum with $m = O(p\ell^2)$ nodes. For convenience, we number the levels of the stratum from 0 to $\ell - 1$, from top to bottom. The stage explores all the nodes in these levels. At any point during the exploration, the set of nodes whose descriptors have been generated but which are not yet

visited is called the *frontier*. (The initial frontier contains all the nodes in the top level of the stratum.) Let $F(j)$ denote those frontier nodes at level j, for $0 \leq j < \ell$ and let $F = \cup_{j=0}^{\ell-1} F(j)$ denote the entire frontier. In order to evaluate the progress that the algorithm is making, we define a *weight* function on the frontier F as follows

$$w(F) = \sum_{j=0}^{\ell-1} |F(j)| 3^{\ell-j} \quad ,$$

i.e. nodes at level j have weight $3^{\ell-j}$. Note that the contribution of a frontier node to $w(F)$ is exponentially decreasing in its level within the stratum. Also, visiting a frontier node at level j involves replacing that node in the frontier by its children (if any), whose combined weight of at most $2 \times 3^{\ell-j-1} = (2/3)3^{\ell-j}$ is a constant factor less than that of their parent. Hence, each node visit decreases the frontier weight. Visiting nodes at lower numbered levels rather than nodes further down the stratum results in a more substantial decrease in the weight function. In order to avoid frequent, expensive balancing steps, our exploration strategy does not necessarily proceed in a regular, breadth-first manner. Nonetheless, we make use of certain cheaper, weight-driven load-balancing techniques to ensure that the frontier weight decreases at a predictable rate. A pictorial representation of the exploration process is shown in Fig. 1

A stage consists of two parts. In the first part, a sequence of (parallel) *visiting steps* is performed to explore nodes in the stratum until the frontier weight is less than or equal to p. In order to detect the end of the first part, visiting

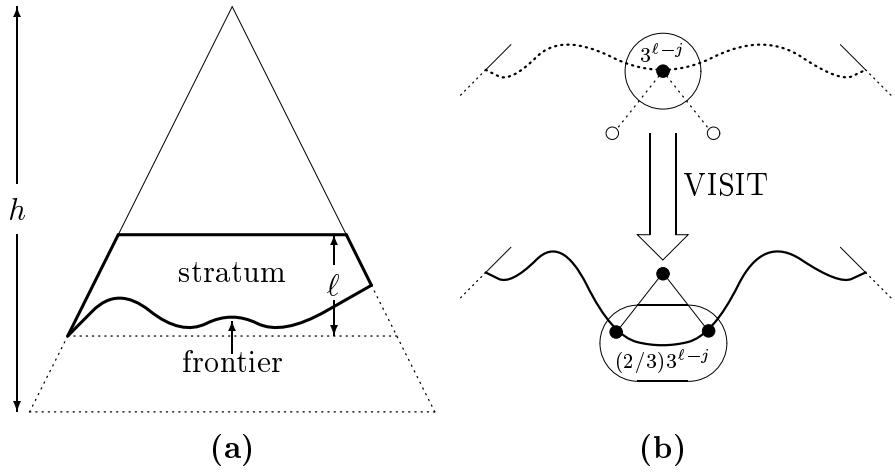

Fig. 1. The weight-driven exploration process on a tree of height h. **(a)** The portion within thin solid lines encloses visited nodes belonging to previous strata. Thick solid lines enclose visited/generated nodes within the current stratum of ℓ levels. Frontier nodes lie along the thick spline. Dashed lines enclose the nodes that are still to be generated. **(b)** The frontier weight reduction induced by a visit of a node at level j of the stratum, with $0 \leq j < \ell$.

steps are executed in batches of ℓ and a weight estimate is computed after the execution of each batch, using the approximate prefix sums algorithm, whose $O(\log\log p)$ complexity is dominated by that of the ℓ visiting steps. The second part of the stage completes the exploration of the stratum as follows. First, for every $0 \leq j < \ell$, a cluster of $2^{\ell-j}$ distinct processors is assigned to each node of $F(j)$ by means of approximate prefix sums in $O(\log\log p)$ time. (Since $\sum_{j=0}^{\ell} |F(j)|2^{\ell-j} \leq \sum_{j=0}^{\ell} |F(j)|3^{\ell-j} \leq p$, such an assignment is feasible.) Next, all of the descendants of each node in F are visited by the corresponding cluster of processors in $O(\ell)$ time. More specifically, consider a frontier node x at level j and let $\{p_0, \ldots, p_{2^{\ell-j}-1}\}$ be the cluster of processors assigned to it. The exploration proceeds in $\ell-1-j$ rounds, where in round k, $0 \leq k < \ell-1-j$, all descendants of x at distance k from it are visited. In round zero p_0 visits x and gives its children (if any) to p_0 and p_1. Thereafter for each round, p_i takes the node (if any) given to it in the previous round, visits it, and gives its children to processors p_{2i} and p_{2i+1}, and so on.

At the end of the stage, the children of the nodes on the last level of the stratum, which make the initial frontier for the next stage, will be evenly distributed among the processors by employing again the approximate prefix sums algorithm.

A very high-level, procedural description of our new strategy for visiting small strata with $O(p\ell^2)$ nodes is given in Fig. 2. In summary, each stratum is visited in a stage (procedure STAGE_VISIT) by first alternating ℓ parallel visiting steps (procedure VISITING_STEP) with an approximate count of the frontier weight (procedure APPROXIMATE_COUNT), until the latter goes below p. Then, the visit of the stratum is completed by first allocating processor clusters to the residual unexplored nodes (procedure ALLOCATE_CLUSTERS), then visiting their subtrees within the stratum using the simple technique illustrated above (procedure COMPLETE_VISIT), and finally redistributing the initial frontier for the next stage to the p processors (procedure REDISTRIBUTE_NODES). Note that the three proce-

```
procedure STAGE_VISIT()
    APPROXIMATE_COUNT(w(F))
    while w(F) > p
        do repeat ℓ times
            VISITING_STEP()
        end repeat
        APPROXIMATE_COUNT(w(F))
    end while
    ALLOCATE_CLUSTERS()
    COMPLETE_VISIT()
    REDISTRIBUTE_NODES()
end STAGE_VISIT
```

Fig. 2. Overall structure of the algorithm for visiting small strata

dures APPROXIMATE_COUNT ALLOCATE_PROCESSORS and REDISTRIBUTE_NODES can all be implemented by means of simple variations of the approximate prefix sums algorithm of [GZ95].

In order to determine the total running time of a stage, we need to give a bound on the number of visiting steps performed. Let F_t be the frontier at the beginning of the tth visiting step. The step is called *full*, if it visits $\Omega(p)$ nodes in F_t, and it is called *reducing* if it visits at least half of the nodes in $\bigcup_{j=0}^{i} F(j)$, for each i in the range $0 \leq i < \ell$. Section 4 will show how to perform a visiting step in time $O((\log\log\log p)^2)$ while ensuring that it is always either full or reducing (see Theorem 10). Clearly, for a stratum of m nodes, there are at most $O(m/p)$ full visiting steps in the stage, whereas the number of reducing steps is bounded by the following lemma.

Lemma 3 *If $m = O(p\ell^2)$, then $O(\ell)$ reducing visiting steps are sufficient to reduce the frontier weight to at most p.*

PROOF. The proof is based on the following property.

Claim. Let $x_0, x_1, \cdots, x_{n-1}$ and $y_0, y_1, \cdots, y_{n-1}$ be two sequences of nonnegative integers such that $\sum_{j=0}^{i} x_j \leq \sum_{j=0}^{i} y_j$, for all $0 \leq i < n$. Then,

$$\sum_{i=0}^{n-1} x_i/3^i \leq \sum_{i=0}^{n-1} y_i/3^i \ .$$

Proof of Claim. The proof is by induction on n. The case $n = 1$ is trivial. Suppose that the property holds for some $n \geq 1$ and consider sequences of $n+1$ elements. Assume that $x_n > y_n$, since otherwise the inductive step is immediate. It is easy to see that

$$\sum_{i=0}^{n} \frac{x_i}{3^i} \leq \sum_{i=0}^{n-2} \frac{x_i}{3^i} + \left(\frac{x_{n-1} + x_n - y_n}{3^{n-1}}\right) + \frac{y_n}{3^n} \ .$$

Note that $\sum_{i=0}^{n-2} x_i + (x_{n-1} + x_n - y_n) \leq \sum_{i=0}^{n-1} y_i$, therefore, by applying the induction hypothesis, we have that

$$\sum_{i=0}^{n-2} \frac{x_i}{3^i} + \left(\frac{x_{n-1} + x_n - y_n}{3^{n-1}}\right) \leq \sum_{i=0}^{n-1} \frac{y_i}{3^i} \ ,$$

which, combined with the previous inequality, proves the claim.

Consider a reducing visiting step. Let F be the frontier prior to the execution of the step and let n_j be the number of nodes in $F(j)$ visited in the step, $0 \leq j < \ell$. Since the visiting step is reducing, we have

$$\sum_{j=0}^{i} n_j \geq \frac{1}{2} \sum_{j=0}^{i} |F(j)| \ ,$$

for any i, $0 \leq i < \ell$, and the claim shows that

$$3^{\ell} \sum_{j=0}^{\ell-1} \frac{n_j}{3^j} \geq \frac{3^{\ell}}{2} \sum_{j=0}^{\ell-1} \frac{|F(j)|}{3^j} = \frac{w(F)}{2}.$$

Thus, the visited nodes account for at least half the total frontier weight. Since the combined weight of the children of any node is at most two thirds of the weight of their parent, it follows that the weight reduction must be at least one third of the total weight of the visited nodes, *i.e.,* at least one sixth of the frontier weight $w(F)$ prior to the execution of the visiting step. Thus, the new frontier weight following the completion of the step is at most $(5/6)w(F)$.

Since the frontier at the beginning of the stage contains $O(p\ell^2)$ nodes at level 0, the initial frontier weight is $O(p\ell^2 3^{\ell})$, which implies that the frontier weight will be less than or equal to p after $O(\ell)$ reducing steps. This proves the lemma.

¿From the above discussion we conclude that our new strategy can be employed to visit any stratum of size $m = O(p\ell^2)$ in $O(m/p + \ell)$ visiting steps and $O((m/p + \ell)(\log\log\log p)^2)$ time. Since strata of size $m = \Omega(p\ell^2)$ can be visited in $O(m/p)$ time using the straightforward breadth-first strategy outlined in Section 2, we can suitably interleave the two strategies and obtain an algorithm that visits any stratum in time $O((m/p + \ell)(\log\log\log p)^2)$ for any value of m. This immediately yields a backtrack search algorithm with the running time stipulated in Theorem 1. Note that the number of visiting steps required is $O(n/p + h)$ in all.

4 Implementation of a Visiting Step

In this section, we describe the implementation of a visiting step which enforces the property that the step is always either full or reducing.

The key idea is a "heap-like" data structure $\mathcal{D}$ that holds the frontier nodes from which nodes are extracted prior to the beginning of the visiting step and to which their children are inserted at the end of that step. Conceptually, $\mathcal{D}$

is composed of an $\ell \times p/\ell$ array of *tree rings*. We also regard the p PRAM processors as being conceptually arranged into ℓ *rows* and $q = p/\ell$ *columns*. At the beginning of the visiting step, the tree rings of the ith row contain all current frontier nodes at level i, $0 \leq i < \ell$. A tree ring is structured as a forest of complete binary trees of different sizes [1]. The leaf vertices in a tree ring are nodes of the tree being visited and each internal vertex contains pointers to its children. The roots of the trees in the same tree ring are organized in a doubly-linked list, ordered by tree size. (This data structure is broadly similar to one used in [CV88].) As in the previous section, we assume that the stratum being visited is of size $O(p\ell^2)$. We use K to denote an upper bound on the size (i.e., the number of node descriptors stored) of any tree ring during the execution of a stage. Later, we will show that $K = O(\ell^3)$, hence the height of any tree in a tree ring will always be $O(\log \ell)$. It should be noted that while each tree ring is notionally associated with a particular processor, since it is stored in the shared PRAM memory it is accessible to all.

A visiting step consists of two sub-steps, VISIT and BALANCE, which are described in the following paragraphs.

VISIT This sub-step is executed in parallel by each column of processors. Let s be the total number of nodes held by the tree rings of the column and let $c > 1$ be a constant to be specified later. The ℓ processors in the column select the $\min\{s, 4c\ell\}$ topmost nodes from the union of their tree rings, and distribute these nodes evenly among themselves. Then, each processor visits the nodes it receives. Finally the children of these just-visited nodes are inserted into the appropriate tree rings within the column.

BALANCE This sub-step is executed in parallel by each row of processors and aims at partially balancing the nodes stored in the tree rings of the row. We define the *degree* of a processor as the number of tree nodes contained in its tree ring. Let f_i be the sum of the degrees of all processors in row i, for $0 \leq i < \ell$. (Note that $f_i = |F(i)|$, *i.e.*, the number of frontier nodes at level i of the stratum.) BALANCE redistributes the nodes among the tree rings in such a way that upon completion at most $\min\{f_i, q\}/(2K)$ processors have degree larger than $c\lceil f_i/q \rceil$ in row i, for any $0 \leq i < \ell$. Moreover, BALANCE never increases the maximum processor degree in any row. The actual implementation of the BALANCE sub-step is rather involved and is discussed separately in Subsection 4.1.

[1] To avoid confusion discussing the elements of the tree being visited and the trees employed in the tree rings, we will use the term *node* exclusively in connection with the former and reserve the term *vertex* for the latter.

We have:

Lemma 4 *A visiting step is always either full or reducing.*

PROOF. Let F be the frontier at the beginning of the visiting step. Then, there are at most $\min\{|F(j)|, q\}/(2K)$ processors in row j of degree larger than $c\lceil |F(j)|/q \rceil$, for each j, $0 \leq j < \ell$. This is ensured either by the BALANCE sub-step executed at the end of the preceding visiting step or, if the visiting step under consideration is the first of the stage, by the (approximately) even distribution of frontier nodes guaranteed at the start of the stage. We call the tree nodes maintained by these overloaded processors *bad nodes* and all the others *good nodes*. Since K is an upper bound to the degree of any processor, we have that the total number of bad nodes in the first i levels of the frontier is

$$K \sum_{j=0}^{i} \frac{\min\{|F(j)|, q\}}{2K} \leq \frac{1}{2} \left| \bigcup_{j=0}^{i} F(j) \right|,$$

for any $0 \leq i < \ell$. Thus the bad nodes at level i or lower account for at most half the total number of frontier nodes at those levels.

Suppose $|F| > 3p$ and let $r \leq q$ be the number of columns holding fewer than ℓ nodes. Since a column holds at most $\sum_{j=0}^{\ell-1} c\lceil F(j)/q \rceil \leq c(|F|/q + \ell)$ good nodes, the number of good nodes is bounded as follows:

$$\frac{|F|}{2} \leq |\{\text{good nodes}\}| \leq r\ell + (q - r)c(|F|/q + \ell),$$

which, following some tedious but simple arithmetic manipulations, implies that $r \leq q(1 - 1/8c)$. Since c is a constant greater than one, we conclude that $q - r \geq q/(8c) = \Theta(q)$ columns hold at least ℓ nodes. Thus, the visiting step will visit $\Theta(q\ell) = \Theta(p)$ nodes, hence the step is full.

Consider now the case $|F| \leq 3p$. Since the number of good nodes in each column is at most $c(|F|/q + \ell) \leq 4c\ell$, it follows that the total number of nodes to be visited in the step is at least equal to the total number of good nodes. From the observation made above, we know that if we visited only the good nodes, then for any $0 \leq i < \ell$ we would visit at least half of the frontier nodes at level i or lower, hence the step would be reducing. Since in each column we select the topmost nodes available, if some good nodes are not visited it can only be because at least the same number of nodes at higher levels are visited in their place, which maintains the reducing property.

In order to implement the visiting step described above, we need efficient primitives to operate on the tree rings. Consider a stage visiting a stratum of size $m = O(p\ell^2)$. Note that at the beginning of the stage the degree of each processor is $O(\ell^2)$, and that after each VISIT sub-step the degree increases by at most an $O(\ell)$ additive term. (Each of the $O(\ell)$ nodes visited by the processors in a column can generate at most two children during a single VISIT step.) Since the BALANCE sub-step does not increase the maximum degree of a processor and $O(m/p + \ell) = O(\ell^2)$ visiting steps are executed overall, we can conclude that the maximum processor degree will always be $O(\ell^3)$. As a consequence, throughout the stage each tree ring contains at most $O(\log \ell)$ trees of $O(\ell^3)$ size and $O(\log \ell)$ height each.

It can be shown [CV88] that:

(1) Given k nodes evenly distributed among $\Theta(k)$ processors, a tree ring whose trees contain these nodes as leaves, can be constructed by the processors in $O(\log k)$ time.
(2) Two tree rings of size $O(k)$ can be merged into one tree ring in $O(\log k)$ time by a single processor.
(3) Any number of k leaves can be extracted by $O(k)$ processors from a tree ring in time proportional to the maximum height of a tree in the tree ring. After extraction, the tree ring structure can be restored within the same time bound.

It can be easily argued that the VISIT sub-step can be implemented in a straightforward fashion within each column using standard techniques such as prefix and the aforementioned primitives to manipulate the tree rings. From the above discussion we conclude:

Lemma 5 *For strata of size* $O(p\ell^2)$, VISIT *can be executed in* $O(\log \ell) = O(\log \log \log p)$ *time.*

4.1 Implementation of BALANCE

As mentioned before, we use $K = O(\ell^3)$ to denote an upper bound to the degree of any processor when a stratum of size $O(p\ell^2)$ is explored (an exact value for K can be derived from the analysis). We assume that K is known by all processors prior to the beginning of the entire algorithm. Since BALANCE is executed in parallel and independently by all rows, we will concentrate on the operations performed by an arbitrary row, say row k. Let f_k denote the total number of tree nodes maintained by the processors of this row at the beginning of the BALANCE sub-step. The purpose of the sub-step is to redistribute these nodes among the processors in such a way that, after the redistribution, the number of processors of degree greater than $c\lceil f_k/q \rceil$ is at

most $\min\{f_k, q\}/(2K)$. (It should be noted that the value f_k is not known to the processors.) The sub-step also ensures that the maximum processor degree is not increased. A crucial feature of the implementation of BALANCE is that nodes are not physically exchanged between the processors, which would be too costly for our purposes, but instead they are "moved" by manipulating the corresponding tree rings, with a cost logarithmic in the number of nodes being moved.

BALANCE is based on a balancing strategy introduced by Broder *et al.* in [BFSU92], which makes use of a special kind of expander defined below.

Definition 6 ([BFSU92]) *An undirected graph $G = (V, E)$ is an (a, b)-*extrovert *graph, for some a, b with $0 < a, b < 1$, if for any set $S \subseteq V$, with $|S| \leq a|V|$, at least $b|S|$ of its vertices have strictly more neighbours in $V - S$ than in S.*

The existence of regular extrovert graphs of constant degree is proved through the probabilistic method in [BFSU92].

For each row, we identify its q processors with the vertices of a regular (a, b)-extrovert graph $G = (V, E)$ of odd degree d, where a, b and d are constants. Let $\gamma = (4d + 3)/(4d + 4)$ and $\tau = \lfloor \log_{1/\gamma} K/2(d + 1) \rfloor$. BALANCE consists of τ *phases*, numbered from 0 to $\tau - 1$. In each phase, some tree nodes maintained by the row processors are marked as *dormant*, and will not participate in subsequent phases. The remaining nodes are said to be *active*. At the beginning of Phase 0 all frontier nodes are active. For $0 \leq i < \tau$, Phase i performs the following actions.

(1) Each processor with more than $K\gamma^i/2$ active nodes in its tree ring declares itself *congested*.

(2) Let $\alpha = 1 + a$ and $\sigma = \lceil \log_{1/(1-b)}(\alpha^\tau 2K) \rceil$. A DAG D is built as a directed version of a subgraph of G. The construction proceeds by performing σ steps of the following type [BFSU92]. Initially, D is empty. In each step, every congested processor not yet in D checks whether at least $(d + 1)/2$ of its neighbours are either non-congested or already in D and, if so, enters D by acquiring edges to $(d + 1)/2$ of these neighbours, which also enter D.
Comment: The construction and the fact that d is odd guarantee that D is a DAG, and that each congested processor in D has out-degree greater than its in-degree, while each non congested processor in the DAG has out-degree 0. Moreover, D has depth at most σ.

(3) A sub-DAG $D' \subseteq D$ is identified comprising all congested processors with more than $K\gamma^{i+1}$ active nodes, and all of their descendants.

(4) Each congested processor not in D that has more than $K\gamma^{i+1}$ active nodes, marks all but $K\gamma^{i+1}$ of its active nodes as dormant.

(5) Let j be such that $2^j \leq K\gamma^i/(2d+2) < 2^{j+1}$. Note that $K\gamma^i/(2d+2) \geq 1$ for $j \leq \tau$. Each processor in D' extracts, for each of its direct successors in D', a tree containing 2^j distinct active nodes from its tree ring, and sends a pointer to this tree to the successor in question.
(6) Each processor merges the trees it receives into its tree ring.

A pictorial representation of the construction of DAGs D and D' performed in a phase of BALANCE is given in Fig. 3.

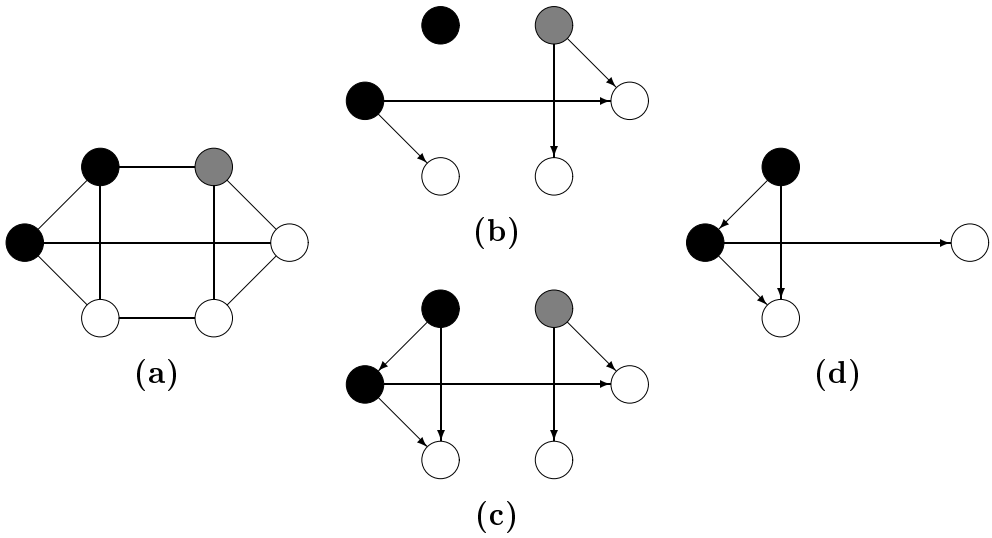

Fig. 3. The DAG construction process performed by BALANCE. **(a)** The extrovert graph G connecting the processors of a row. White nodes represent uncongested processors. Black and shaded nodes represent, respectively, "heavily" congested processors (more than $K\gamma^{i+1}$ active nodes) and "lightly" congested processors (more than $K\gamma^i/2$ and at most $K\gamma^{i+1}$ active nodes). **(b)-(c)** Two-step construction of DAG D. **(d)** The final subdag $D' \subseteq D$ containing all heavily congested nodes and their descendants in D.

In what follows, we show that at the end of the τ phases the number of processors of degree more than $c\lceil f_k/q \rceil$ is at most $\min\{f_k, q\}/(2K)$, and that the maximum processor degree is not increased.

Lemma 7 *For $0 \leq i < \tau$, at the beginning of Phase i each processor holds at most $K\gamma^i$ active nodes and at most $K(1-\gamma^i)$ dormant nodes. Moreover, no phase increases the maximum processor degree.*

PROOF. We proceed by induction on i. The case $i = 0$ is clearly true. Suppose that the property holds up to index i. By induction, each processor starts Phase i with at most $K\gamma^i$ active nodes and at most K nodes overall. A congested processor that does not make it into the DAG D is not involved in any movement of nodes in the phase. Each such processor begins with at most K nodes ($K\gamma^i$ active and $K(1-\gamma^i)$ dormant) and at the end of the phase

at most $K\gamma^{i+1}$ of its nodes remain active while the rest become (or remain) dormant. Moreover, the processor's degree does not change. If sub-DAG D' is empty then all congested processors in D have at most $K\gamma^{i+1}$ active nodes and at most $K - K\gamma^{i+1} = K(1-\gamma^{i+1})$ dormant nodes. Since in this case no exchange of pointers takes place, the property for index $i+1$ follows. Suppose instead that D' is not empty, that is, there is at least one congested processor in D with more than $K\gamma^{i+1}$ active nodes. (Note that in this case the maximum processor degree is greater than $K\gamma^{i+1}$). A congested processor in D' transmits 2^j active nodes to each of its successors. Since the out-degree of a congested processor in D' is greater than its in-degree this represents a net loss of at least $2^j \geq K\gamma^i/(4d+4)$ nodes. Therefore, in any such processor the number of active nodes at the end of the phase is at most

$$K\gamma^i - \frac{K\gamma^i}{4d+4} = K\gamma^{i+1},$$

and its overall degree is decreased. Moreover, the number of dormant nodes for such a processor stays unchanged, namely $K(1-\gamma^i) \leq K(1-\gamma^{i+1})$. Finally, a non-congested processor begins the phase with at most $K\gamma^i/2$ active nodes and receives at most $d2^j \leq dK\gamma^i/(2d+2)$ new active nodes, which adds up to

$$\frac{K\gamma^i}{2} + \frac{dK\gamma^i}{2d+2} \leq K\gamma^{i+1},$$

which is less than the maximum processor degree. As in the previous case, the number of dormant nodes for such a processor stays unchanged, that is less than $K(1-\gamma^{i+1})$.

We refer to the processors maintaining dormant nodes as *rogues*. Let $R(j)$ denote the set of rogues at the beginning of Phase j and $C(j)$ the set of processors that declare themselves congested in the phase. Define $r_j = |R(j)|$ and $c_j = |C(j)|$, for $0 \leq j < \tau$. Let $\tau' = \left\lfloor \log_{1/\gamma}\left(\frac{Ka}{2\lceil f_k/q \rceil}\right)\right\rfloor$. We have:

Lemma 8 *For $0 \leq j \leq \min\{\tau', \tau\}$, we have*

$$r_j \leq \alpha^j (1-b)^\sigma \min\{f_k, q\} \ .$$

PROOF. We proceed by induction on j. The case $j = 0$ is clearly true since $r_0 = 0$. Suppose that the property holds up to index $j-1$ and consider index j. Note that the rogues at the beginning of Phase j will be given by the set $R(j-1)$ plus a set $C' \subseteq C(j-1)$ containing congested processors that did not make it into the DAG during Phase $j-1$. Let us give an upper bound to

$|C'|$. Note that $c_{j-1} \leq a \min\{f_k, q\} \leq aq$, since otherwise congested processors would account for more than

$$\frac{K\gamma^{j-1} a \min\{f_k, q\}}{2} > \frac{K\gamma^{\tau'} a \min\{f_k, q\}}{2} \geq f_k$$

active nodes, which is impossible. By the extrovertness of the graph G, after the first t steps of DAG construction, the number of congested processors not in D are at most $c_{j-1}(1-b)^t$. This implies that $|C'| \leq c_{j-1}(1-b)^\sigma$, hence the number of rogues at the beginning of Phase j will be

$$\begin{aligned} r_j &\leq c_{j-1}(1-b)^\sigma + r_{j-1} \\ &\leq a(1-b)^\sigma \min\{f_k, q\} + \alpha^{j-1}(1-b)^\sigma \min\{f_k, q\} \quad \text{(by induction)} \\ &\leq \alpha^j (1-b)^\sigma \min\{f_k, q\} \ . \end{aligned}$$

Lemma 9 *By the end of the BALANCE procedure, the number of processors of degree more than $c\lceil f_k/q \rceil$ is at most $\min\{f_k, q\}/(2K)$ for a suitable choice of the constant c. Moreover, the procedure is executed in time $O((\log\log\log p)^2)$ on the COMMON CRCW-PRAM.*

PROOF. Let us first consider the case $\tau' \leq \tau$. At the beginning of Phase τ', each processor maintains at most $K\gamma^{\tau'} \leq c\lceil f_k/q \rceil$ active nodes (by Lemma 7 provided $c \geq 2/a\gamma$), and the number of rogues is

$$r_{\tau'} \leq \alpha^{\tau'}(1-b)^\sigma \min\{f_k, q\} \leq \frac{\min\{f_k, q\}}{2K} \tag{1}$$

(by Lemma 8 and the choice of σ). Moreover Lemma 7 shows that the maximum degree of processors that are not rogues at the end of Phase τ' will not increase above the $c\lceil f_k/q \rceil$ threshold in the subsequent $\tau - \tau' - 1$ phases.

Now consider the case where $\tau < \tau'$. In this case

$$K\gamma^\tau \leq \frac{2(d+1)}{\gamma} \leq c\lceil f_k/q \rceil$$

for $c \geq 2(d+1)/\gamma$. Moreover, since r_j is increasing in j, the total number of rogues is no more than that indicated in Equation 1.

We now evaluate the running time. Consider a phase of BALANCE. Step 1 clearly takes $O(1)$ time. Every DAG construction step is accomplished in constant time, hence the construction of D (Step 2) takes time $O(\sigma)$. Since D has depth at most σ it is easy to see that Step 3 can be accomplished in time

$O(\sigma)$, as well. The cost of the remaining steps is dominated by the cost of the extraction and merging operations performed on the tree rings, which take $O(\log K)$ time overall. Noting that the number of phases is $\tau = O(\log K)$ and $\sigma = O(\tau + \log K) = O(\log K)$, we conclude that the overall running time is

$$O(\tau \log K) = O(\log^2 K) = O(\log^2 \ell) = O((\log \log \log p)^2).$$

The following theorem combines the contributions of this section and establishes the result announced in Section 3 upon which the analysis of our backtrack strategy is based.

Theorem 10 *A visiting step within a stratum of size $O(p\ell^2)$ can be implemented in $O((\log \log \log p)^2)$ time on a p-processor COMMON CRCW-PRAM, while ensuring that the step is either full or reducing.*

5 Evaluation of Bounded-Degree DAGs

In this section we show how some of the ideas involved in the backtrack search algorithm may be used to solve the DAG evaluation problem. In a *computation DAG*, nodes with zero in-degree are regarded as *inputs*, while other nodes represent operators whose operands are the values computed by their predecessors (*i.e.*, nodes adjacent with respect to incoming edges). Nodes with zero out-degree are regarded as *outputs*. A node can be evaluated only after all of its operands have been evaluated. The *DAG evaluation problem* consists of evaluating all output nodes. We define the layers of the DAG in the obvious way: the inputs are at layer zero and the layer of every other node is one plus the maximum layer among its predecessors.

In our parallel setting, we assume that a DAG D of constant degree is stored in the shared memory of a p-processor COMMON-CRCW PRAM. Each DAG node is represented by a *descriptor* containing the following information: a field that specifies the type of operation associated to that node; a field to store its value; two fields for each operand (*i.e.*, each incoming edge), where processors will write the value of the operand, and a timestamp to record the time of writing; and pointers to the node's successors in D. Initially, only pointers to the descriptors of the DAG inputs are known and evenly distributed among the processors.

Notice the similarity between the DAG evaluation and the backtrack search problems. While the computational DAG is not necessarily a tree, nevertheless we can still visit (*i.e.*, evaluate) it by proceeding in a quasi breadth-first stratum-by-stratum fashion as in the backtrack search algorithm.

More precisely, recall that in the backtrack search problem a node is revealed by the processor that visits its (unique) parent. In the DAG evaluation problem, "visiting" a node entails computing the node's value and writing this value in the node's descriptor and, together with a timestamp, in the appropriate fields of its successors' descriptors. A node is revealed (hence ready to be evaluated/visited itself) only when the last of its predecessors has been evaluated, and the node is regarded as being a "child" of that predecessor (with ties being broken arbitrarily). In this fashion, a spanning forest for the DAG is implicitly identified and the DAG evaluation can be regarded as a visit of this a forest.

By noting that our backtrack search algorithm can be employed to visit any forest of bounded-degree trees in $O((n/p + h)(\log\log\log p)^2)$ time, where n is the total number of nodes and h the maximum tree-height in the forest, we conclude that the DAG evaluation problem can be solved within the same time bound.

6 Conclusions

In this work we devised an efficient deterministic strategy for performing parallel backtrack search on a shared memory machine. Specifically, our strategy attains a running time which is only a triply logarithmic factor away from a natural lower bound for the problem. As with all previous studies, our investigation has mainly focused on running time. On the other hand, the overall space required by our algorithm can grow as large as the tree size n, whereas the space required by the randomized schemes proposed in [KZ93,LAB93,Ran94] is bounded above by $\min\{n, ph\}$. This latter quantity, however, is close to n for large values of p and/or highly unbalanced trees. It remains a challenging open problem to devise fast and space efficient backtrack search algorithms and, more generally, to study time-space tradeoffs for parallel backtrack search.

References

[AL91] B. Aiello and T. Leighton. Coding theory, hypercube embeddings and fault-tolerance. In *Proceedings of 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 125–136, 1991.

[BFSU92] A.Z. Broder, A.M. Frieze, E. Shamir, and E. Upfal. Near-perfect token distribution. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, Volume 623, pages 308–317, July 1992, Springer-Verlag.

[BGLL91] S.N. Bhatt, D. Greenberg, F.T. Leighton, and P. Liu. Tight bounds for on-line tree embeddings. In *Proceedings of the 2nd ACM-SIAM Symposium On Discrete Algorithms*, pages 344–350, January 1991.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17(1):128–142, 1988.

[GZ95] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Proceedings of the 4th Israel Symposium on Theory of Computing and Systems*, pages 220–228, 1995.

[HPP99a] K. Herley and A. Pietracaprina and G. Pucci. Fast deterministic parallel branch and bound. *Parallel Processing Letters*, 9(3):325–334, 1999.

[HPP99b] K. Herley and A. Pietracaprina and G. Pucci. Deterministic branch and bound on distributed-memory machines. *International Journal on Foundations of Computer Science*, 10(4):391–404, 1999.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, Reading MA, 1992.

[KP94] C. Kaklamanis and G. Persiano. Branch-and-bound and backtrack search on mesh-connected arrays of processors. *Mathematical Systems Theory*, 27:471–489, 1994.

[KZ93] R.M. Karp and Y. Zhang. parallel algorithms for backtrack search and branch and bound computation. *Journal of the ACM*, 40:765–789, 1993.

[LAB93] P. Liu, W. Aeillo and S. Bhatt. An atomic model for message-passing. In *Proceedings of 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 154–163, 1993.

[LNRS92] T. Leighton, M. Newman, A. G. Ranade and E. Schwabe. Dynamic tree embeddings in butterflies and hypercubes. *SIAM Journal on Computing*, 21:639–654, 1992.

[Ran90] A.G. Ranade. A simpler analysis of the Karp-Zhang parallel branch-and-bound method. Technical Report No. 586, Computer Science Division, University of California at Berkeley, Berkeley, California, 1990.

[Ran94] A.G. Ranade. Optimal speed-up for backtrack search on a butterfly network. *Mathematical Systems Theory*, 27:85–101, 1994.

[WK91] I.C. Wu and H.T. Kung. Communication complexity for parallel divide-and-conquer. In *Proc. of the 32nd IEEE Symp. on Foundations of Computer Science*, pages 151–162, 1991.