DeltaTree: A Locality-aware Concurrent Search Tree

Ibrahim Umar
Department of Computer Science
University of Tromsø, Norway
ibrahim.umar@uit.no

Otto J. Anshus
Department of Computer Science
University of Tromsø, Norway
otto@cs.uit.no

Phuong Hoai Ha
Department of Computer Science
University of Tromsø, Norway
phuong@cs.uit.no

ABSTRACT

Like other fundamental abstractions for high-performance computing, search trees need to support both high concurrency and data locality. However, existing locality-aware search trees based on the van Emde Boas layout (vEB-based trees), poorly support concurrent (update) operations.

We present DeltaTree, a practical locality-aware concurrent search tree that integrates both locality-optimization techniques from vEB-based trees, and concurrency-optimization techniques from highly-concurrent search trees. As a result, DeltaTree minimizes data transfer from memory to CPU and supports high concurrency. Our experimental evaluation shows that DeltaTree is up to 50% faster than highly concurrent B-trees on a commodity Intel high performance computing (HPC) platform and up to 65% faster on a commodity ARM embedded platform.

Categories and Subject Descriptors

D.1.3 [Concurrent programming]: Parallel programming;
D.4.8 [Performance]: Measurements—performance evaluation, concurrent performance; G.1.0 [General]: Parallel algorithms—complexity measures, performance measures

Keywords

Performance evaluation; concurrent algorithms; data locality; multi-core processors; memory systems

1. INTRODUCTION

The conventional van Emde Boas (vEB) layout based trees are examples of locality-aware search trees found in several research on cache-oblivious (CO) data structure [1–5].

The main feature of the vEB layout is that the cost of any search is $O(\log_B N)$ memory transfers, where $N$ is the tree size and $B$ is the unknown memory block size in the CO model [2]. As such, its search is cache-oblivious. The search cost is optimal and matches that of B-trees. However, B-trees

2. RELAXED CO MODEL AND CONCURRENCY-AWARE VEB LAYOUT

In order to make the vEB layout suitable for highly concurrent data structures with concurrent update operations, we introduce a novel concurrency-aware dynamic vEB layout. Our key idea is that if we know an upper bound $UB$ on the unknown memory block size $B$, we can support dynamic node allocation via pointers while maintaining the optimal search cost of $O(\log_B N)$ memory transfers without knowing $B$ (cf. Lemma 2.1). This idea is based on that in practice it is not feasible to keep the vEB layout in a contiguous block of physical memory greater than some upper bound set by the underlying system physical page size (frame size) and cache-line size.

Figure 1 illustrates the new concurrency-aware vEB layout based on the relaxed CO model. The memory transfer cost for search operations in the new concurrency-aware vEB layout is the same as that of the conventional vEB layout (cf. [6]).
Lemma 2.1. However, the concurrency-aware vEB supports high concurrency for update operations.

We define relaxed cache-oblivious algorithms as cache-oblivious (CO) algorithms with the restriction that an upper bound UB on the unknown memory block size B is known in advance. As long as an upper bound on all the block sizes of multilevel memory is known, the new relaxed CO model maintains the key features of the original CO model [5]. This enables algorithm designs that can utilize fine-grained data locality in the multilevel memory hierarchy of modern architectures.

**Lemma 2.1.** For any upper bound UB on the unknown memory block size B, a search in a complete binary tree with the concurrency-aware vEB layout achieves the optimal memory transfer O(log
\(_B\) N), where N and B are the tree size and the unknown memory block size in the CO model, respectively.

**Proof.** (Sketch) Figure 1b illustrates the proof. Let k be the coarsest level of detail such that every recursive subtree contains at most B nodes. Since B \(\leq UB\), k \(\leq L\), where L is the coarsest level of detail at which every recursive subtree (∆Nodes) contains at most UB nodes. Consequently, there are at most \(2^{L-k}\) subtrees along the search path in a ∆Node and no subtree of depth \(2^{k}\) is split due to the boundary of ∆Nodes. Namely, triangles of height \(2^{k}\) fit within a dashed triangle of height \(2^{L}\) in Figure 1b.

Because at any level of detail \(i \leq L\) in the concurrency-aware vEB layout, a recursive subtree of depth \(2^{k}\) is stored in a contiguous block of memory, each subtree of depth \(2^{k}\) within a ∆Node is stored in at most two memory blocks of size B (depending on the starting location of the subtree in memory). Since every subtree of depth \(2^{k}\) fits in a ∆Node (i.e., no subtree is stored across two ∆Nodes), every subtree of depth \(2^{k}\) is stored in at most two memory blocks of size B.

Since the tree has height \(T\), \([T/2^h]\) subtrees of depth \(2^k\) are traversed in a search and thereby at most 2\([T/2^h]\) memory blocks are transferred.

Since a subtree of height \(2^{k+1}\) contains more than B nodes, \(2^{k+1} \geq \log_B (B + 1)\), or \(2^k \geq \frac{1}{2} \log_B (B + 1)\).

We have \(2^{T-1} \leq N \leq 2^T\) since the tree is a complete binary tree. This implies \(\log_B N \leq T \leq \log_B N + 1\).

Therefore, the number of memory blocks transferred in a search is \(2\left[\frac{T}{2^h}\right] \leq 4\left[\log_B N + \log_B(B + 1)\right] = O(\log_B N)\), where \(N \geq 2\). \(\square\)

4. ACKNOWLEDGMENTS

This research work has received funding from the European Union Seventh Framework Programme (grant n°611183) and from the Research Council of Norway (grant n°231746/F20). The authors would like to thank the Norwegian Metacentrum for Computational Science for giving us access to HPC clusters.

5. References


Using 5 million operations in two micro-benchmarks: 100% search and 5% update (95% search) using random numbers. 