View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KITopen

Master's Thesis

Engineering Parallel Bi-Criteria Shortest Path Search

Stephan Erb

June 19, 2013

Advisors: Prof. Dr. Peter Sanders
Dipl. Inform. Moritz Kobitzsch

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Name:	Stephan Erb
Student ID:	1609867
Pursued Degree:	Master of Science
Date of Submission:	June 19, 2013
Editing Time:	6 Months

Acknowledgments:

I would like to thank my family and friends for their patience and support, my colleagues for the fruitful discussions, and the numerous reviewers of this thesis for their invaluable suggestions helping to improve my writing.

Furthermore, I would like to thank Raith and Ehrgott [54] and Machuca and Mandow [36] for providing us with their road networks, and Dennis Schieferdecker for the generation of the sensor networks.

Contact Information:

Author:
Stephan Erb
E-Mail: thesis@stephanerb.eu
`http://stephanerb.eu`

University:
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
Phone: +49 721 608-0
Fax: +49 721 608-44290
E-Mail: info@kit.edu
`http://www.kit.edu`

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 19. Juni 2013

Abstract

The multi-criteria shortest path problem is concerned with the minimization of multiple, possibly conflicting objectives such as travel time and highway tolls. Sanders and Mandow [60] present a parallel algorithm for this problem. They generalize Dijkstra's [18] single-criterion shortest path algorithm to multiple criteria, while parallelizing all additional work induced by this generalization. For the bi-criteria case they even show that their algorithm performs less work than classic bi-criteria algorithms.

We discuss in how far these theoretical results translate into an efficient implementation for modern shared-memory multiprocessors. In particular, we focus on a cache-efficient implementation of the bi-criteria case and present an extensive evaluation of the algorithm and its underlying data structures.

The evaluation indicates that parallel bi-criteria shortest path search is feasible in practice. Our implementation exhibits significant speedups when solving difficult problem instances in parallel. In addition, even a sequential implementation is competitive to classic bi-criteria shortest path algorithms.

As an additional contribution of independent interest, we present parallel bulk updates for weight-balanced B-trees [3]. We use this technique to implement the Pareto queue, a multi-dimensional generalization of a priority queue proposed by Sanders and Mandow [60] and required by their algorithm.

Zusammenfassung

Bei der Berechnung von kürzesten Wegen gibt es häufig nicht nur ein, sondern mehrere interessante Kriterien, wie zum Beispiel die Reisezeit und die Mautkosten für die jeweilige Strecke. Sanders und Mandow [60] präsentieren einen parallel Algorithmus für dieses multi-kriterielle kürzesten Wege Problem. Bei dem Algorithmus handelt es sich um eine Verallgemeinerung des Algorithmus von Dijkstra [18] von einem auf mehrere Kriterien, jedoch unter Parallelisierung jeglicher zusätzlich entstehender Arbeit. Für den bi-kriteriellen Fall kann sogar gezeigt werden, dass im parallelen Algorithmus weniger Arbeit als in klassischen Ansätzen anfällt.

In der vorliegenden Arbeit wird untersucht, inwieweit sich diese theoretischen Ergebnisse in die Praxis übertragen lassen. Wir erarbeiten hierzu eine cache-effiziente, bi-kriterielle Implementierung des Algorithmus für moderne Multiprozessorsysteme mit geteiltem Speicher.

Unsere experimentellen Untersuchungen zeigen die Praktikabilität paralleler bi-kriterieller Suche. Die parallele Implementierung des Algorithmus liefert signifikante Beschleunigungen für Probleminstanzen mit unkorrelierten oder negativ korrelierten Kriterien. Darüber hinaus kann für diese Instanzen sogar gezeigt werden, dass selbst im sequentiellen Fall der parallele Algorithmus kompetitiv zu klassischen bi-kriteriellen Verfahren ist.

Als zusätzlicher Beitrag allgemeineren Interesses präsentieren wir parallele Massenupdates für gewichtsbalancierte B-Bäume [3]. Wir verwenden diese Technik zur Realisierung der Pareto queue, einer mehrdimensionalen Verallgemeinerung von Prioritätswarteschlangen, welche von Sanders und Mandow [60] im Zuge ihres parallelen Algorithmus eingeführt wurde.

Contents

List of Figures

List of Algorithms

List of Tables

1. Introduction

The problem of finding the quickest route from one location to another, the shortest path problem, is ubiquitous in our daily lives. Classic algorithms solving it, such as Dijkstra's [18] algorithm, are designed to minimize a *single* metric such as travel time. Unfortunately, this simple model is not sufficient for all interesting shortest path problems. For example, we may be interested in the simultaneous optimization of multiple, conflicting objectives such as travel time and highway tolls. In these cases, there may not be only one optimal route, but several reasonable ones, all with their own advantages and disadvantages: a fast route with high costs, a slightly slower route but with slightly lower costs, etc. Given an algorithm that computes all these alternative routes for us, we can select a route according to our personal preferences, e.g., depending on how much we value our time.

The shortest path problem concerned with minimizing *multiple* objectives is called the *multi-criteria shortest path problem.* As indicated in the example given above, it deals with the computation of all optimal alternative routes, the so-called *Pareto optimal* paths. It has a large number of applications, including but not limited to vehicle routing [30], routing in railroad networks [44] and routing in multimedia networks [15].

Even the shortest path problem with only two criteria, the *bi-criteria shortest path problem*, is NP hard [20]. The crucial parameter determining its complexity is the total number of Pareto optima [44], which can be exponential in the number of nodes [27]. Even though many real world applications do not exhibit these worst-case properties [44, 39], solving them still requires significant resources. Guerriero and Musmanno [23] therefore conclude that parallel computing is the main goal for future development in this field. By using today's hardware more efficiently, parallel shortest path search promises faster computation of existing problem instances and may make the computation of even larger instances feasible.

Sanders and Mandow [60] present a parallel label-setting algorithm for the multi-criteria shortest path problem. The algorithm is a generalization of Dijkstra's [18] algorithm for the multi-criteria case and relies on a multi-dimensional generalization of a priority queue, the *Pareto queue.* To our knowledge, it is the first proposal of a parallel multi-criteria shortest path algorithm, which however, has not been implemented yet.

1.1. Problem Statement

The *one-to-all bi-criteria shortest path problem* is concerned with finding all Pareto optimal paths from a single source node to all other nodes of a given graph. Our goal is to investigate if this problem can be efficiently solved in parallel. More specifically, we want to evaluate whether a careful, bi-criteria implementation of the algorithm of Sanders and Mandow [60] can be efficient in solving large inputs on modern shared-memory multiprocessors.

In contrast to classic sequential bi-criteria shortest path algorithms, the parallel algorithm explores several Pareto optimal paths in parallel. Identifying and exploring these paths comes with additional overhead, but may be worthwhile if, depending on the problem instance, many paths can be explored simultaneously. Unfortunately, Sanders and Mandow [60] point out that their algorithm may be too complicated to be efficient in practice, as they adopted fine-grained parallelism to achieve good asymptotic results.

Most modern server and desktop systems are equipped with shared-memory multiprocessors. Given their quick inter-processor communication, they are an ideal target for our implementation effort. However, multiprocessors are diverging from traditional machine models used for algorithm design [59]. Given the potential inefficiency of the algorithm, we therefore have to incorporate knowledge about the hardware in order to devise a fast implementation. For example, we have to consider memory hierarchies, issues like false sharing, and in general, constant factors instead of plain asymptotics.

Even if our parallel implementation does not achieve perfect speedups over classic sequential competitors, we might still be able to show that parallelization can be beneficial for certain problem instances.

1.2. Contributions

The contributions of this thesis (a short summary intended for informed readers):

- *Parallel Bi-Criteria Shortest Path Search:* We present an implementation of the algorithm of Sanders and Mandow [60] based on cache-efficient B-trees. We use this implementation to validate the practicality of parallel multi-criteria shortest path search for synthetic and real world instances. We make two major observations:
  - A serial execution of the algorithm is competitive to classic sequential bi-criteria shortest path algorithms, at least for problem instances with uncorrelated and negatively correlated objectives. For the latter case, the serial execution can even be slightly faster than its classic competitor.
  - The parallel algorithm is practical on modern shared-memory multiprocessors for sufficiently difficult problem instances, achieving absolute speedups of at least three for four threads and five for eight threads. For larger number of threads, the speedup of our implementation is bound by the available memory bandwidth.
- *Bulk Updates for B-tree:* We contribute a technique for the parallel bulk update of weight-balanced B-trees [3] and use it as the basis for a cache-efficient implementation of the parallel bi-criteria algorithm. Given p threads and a tree of size N, our technique has an amortized update bound of $\mathcal{O}(k/p \cdot \log N)$ parallel time for the application of a sequence of k updates. The same runtime bound is achieved by the binary tree-based competitor algorithm of Frias and Singler [19]. However, we are able to show that even a sequential implementation of our technique is competitive to

this parallel competitor using eight threads. In addition, our parallel implementation achieves an absolute speedup of at least three for four threads and six for eight threads.

- *Classic Bi-Criteria Shortest Path Search:* We contribute to the research on sequential bi-criteria shortest path algorithms by engineering a classic competitor for our parallel implementation. We show how label selection rules [47, 29] can affect the access pattern on label sets. By exploiting these findings, our implementation is able to outperform label setting and label correcting implementations presented in recent literature (i.e., in [54]).

1.3. Outline

This thesis starts with a discussion of the bi-criteria shortest path problem and with classic algorithms solving it. Based on this background information, we present the parallel algorithm of Sanders and Mandow [60] as proposed by its authors.

The parallel algorithm relies on batched updates of balanced search trees. In Chapter 3, we therefore investigate (parallel) bulk updates for cache-efficient trees. After a short review of the relevant literature, we present our own solution, including its analysis and evaluation.

In Chapter 4, we detail the implementation of the parallel algorithm of Sanders and Mandow [60] based on our cache-efficient tree. We also provide the remaining details of our implementation, including means to reduce the need for synchronization and load balancing. Our effort results in an efficient shared-memory implementation.

The implementation is evaluated in Chapter 5. We start with the development of a tuned reference implementation of a classic shortest path algorithm, serving as a competitor for the parallel implementation. The evaluation is then carried out for a combination of synthetic and realistic road, grid and sensor networks. We conclude our work in Chapter 6 with a short retrospective and several proposals for future work.

2. Bi-Criteria Shortest Path Search

We look into basic concepts and classic algorithms for bi-criteria shortest path search. Based on this knowledge, we then present the parallel algorithm of Sanders and Mandow [60].

2.1. Definitions and Terminology

We introduce basic terminology in order to define and explain the *one-to-all bi-criteria shortest path problem.*

Definition 2.1 (Graph, Weight, Path, Length)**.** Let $G = (V, E)$ be a directed *graph* with n nodes and m edges. Each edge of this graph has a *weight* in form of a two-dimensional cost vector $\vec{w} = (x, y)$. The costs x and y are non-negative integer values (e.g., the time and cost required to traverse an edge). A *path* in G is a sequence of edges. The *length* of a path is defined as the component-wise sum of all of its edge weights. For example, a path p consisting of three edges with weights $\vec{w_1} = (10, 7)$, $\vec{w_2} = (1, 3)$, and $\vec{w_3} = (2, 2)$ has the length $l = (13, 12)$. Given a path p, we refer to the first cost component of its length as p_x and to the second as p_y (i.e., in our example $p_x = 13$ and $p_y = 12$).

To decide which paths are better than others, we adopt the concept of *Pareto optimality*:

Definition 2.2 (Pareto optimality, Dominance, Pareto front)**.** A path p from node u to node v is called *Pareto optimal* if it is not *dominated* by any other path from u to v. A path p is *dominated* by a path q if and only if $(q \neq p) \wedge (q_x \leq p_x) \wedge (q_y \leq p_y)$. This means, p is *dominated* if q is better or equal to p in both dimensions and strictly better in at least one of them. Given a set of paths, the Pareto optimal subset is called the *Pareto front.*

Figure 2.1 explains this concept using a geometrical representation of our two-dimensional objective space: Given a point p, we draw a shaded region marking all coordinates in both dimensions that are larger than the coordinates of p. A point is dominated if it falls into the shaded region of another point. Pareto optimal points lie outside of all shaded regions and are therefore not dominated.

We can finally define the shortest path problem subject to this thesis.

Definition 2.3 (Bi-criteria shortest path problem)**.** The *one-to-all bi-criteria shortest path problem* is concerned with finding all Pareto optimal paths from a single source node to all other nodes in a given graph G.

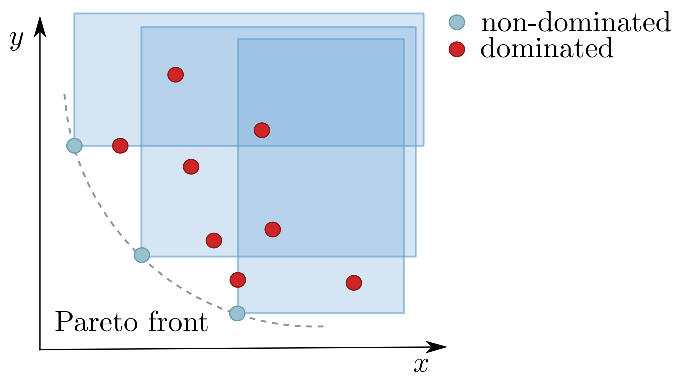


Figure 2.1.: Dominated and non-dominated (i.e., Pareto optimal) solutions within the bi-dimensional objective space.

However, we are only interested in computing *minimal complete sets* of Pareto optimal paths (e.g., see [29]): Even though Pareto optimality does not exclude several paths to the same node with the same length, we only want to compute one of these paths.

Before we look into classic algorithms solving this problem, we have to introduce an important ordering of paths, the *lexicographic order*. The lexicographic order has a few important properties (at least) in the bi-criteria case, which we summarize in form of several observations.

Definition 2.4 (Lexicographic order)**.** A path p is *lexicographically* smaller than a path q if and only if $(p_x < q_x) \vee (p_x = q_x \wedge p_y < q_y)$

Observation 2.1 (Identification of *one* Pareto optimum)**.** Given a set of paths, the lexicographically smallest one is always Pareto optimal. For example, of all dominated and non-dominated points in Figure 2.1, the left-most one is Pareto optimal.

Observation 2.2 (Identification of *all* Pareto optima)**.** The Pareto optimal subset of a set of lexicographically sorted paths is equivalent to the prefix minima over y-coordinates of paths [60]: A point in Figure 2.1 is Pareto optimal if its y-coordinate is smaller than the y-coordinates of all lexicographically preceding points (i.e., points with smaller x-coordinate).

Observation 2.3 (Sorting of Pareto optima)**.** If we sort Pareto optima (e.g., all non-dominated points in Figure 2.1) by their increasing x-coordinate, they are also sorted by their decreasing y-coordinate.

As a preparation for presentation of the parallel shortest path algorithm, we introduce the *task* concept:

Definition 2.5 (Task, Work stealing)**.** A *task* is an object, which represents work of the application to be performed in parallel. Tasks are executed by threads, each of which maintains a work list of scheduled tasks. If a thread has finished all tasks in his local work list, it attempts to steal tasks from other threads using a randomized *work-stealing algorithm [8, 9].*

Tasks allow us to logically decompose a problem without having to worry about available hardware cores or threads. We call a task *recursive* if it behaves like a recursive procedure, i.e., it either reaches a base case or spawns additional tasks of the same type.

2.2. Classic Algorithms

Generally speaking, multi-criteria shortest path algorithms can be classified as either exact, heuristic or approximate [21]. Exact algorithms can further be subdivided into labeling algorithms and ranking algorithms. We restrict our following discussion to exact labeling algorithms, as they are closest to the approach of Sanders and Mandow [60]. For details on other approaches, we refer to recent surveys of multi-criteria [21] and bi-criteria [64, 54] algorithms.

Labeling algorithms grow shortest paths by computing and assigning *labels* to nodes, similar to Dijkstra's [18] algorithm. However, instead of just one label, multi-criteria labeling algorithms have to maintain a set of labels for each node.

Definition 2.6 (Label (permanent, tentative))**.** A *label* represents a path from the start node to the respectively labeled node and encodes the length of this path. Representing paths, Definitions 2.2 and 2.4 also apply to labels, i.e., labels can dominate each other, non-dominated labels are called Pareto optimal, and labels can be sorted lexicographically. A *permanent* label denotes a path known to be Pareto optimal. *Tentative* labels denote temporary results not yet known to be Pareto optimal.

Labeling algorithms are either *label setting* or *label correcting* and differ in when and how tentative labels become permanent. We proceed to describe their common core, before detailing their differences.

Definition 2.7 (Labeling Algorithm Data Structures (Queue, Label Set))**.** Labeling algorithms rely on the following abstract data types:

- *Queue Q:* A set of tentative labels serving as the work list of the algorithm. A *label selection strategy* controls the order in which tentative labels are removed from the queue, determining which paths are explored next. Even though we refer to the elements in Q as labels, they are in fact pairs of node ID and label, in order to track the assignment of labels to nodes.
- *Label Set $L[v]$ for each $v \in V$:* The collection of permanent and tentative labels assigned to node v. Only non-dominated labels which are not equal to an existing label can be inserted into this set, a policy enforced by a *dominance check* which is also responsible to delete existing labels dominated by newly inserted ones.

Algorithm 2.1: Classic Bi-criteria Labeling Algorithm

```
Input: Graph $G = (V, E)$, start node $s$
Output: L
$L[v] \leftarrow \emptyset$ for all $v \in V$
Queue $Q \leftarrow \{(s, (0, 0))\}$
while $Q$ is not empty do
    $(u, l) \leftarrow Q.nextLabel()$
    foreach $(u, v) \in E$ do
        $w \leftarrow weight((u, v))$
        $l' \leftarrow (l_x + w_x, l_y + w_y)$
        if $l'$ is not dominated by $L[v]$ then
            update $L[v]$
            update $Q$
```

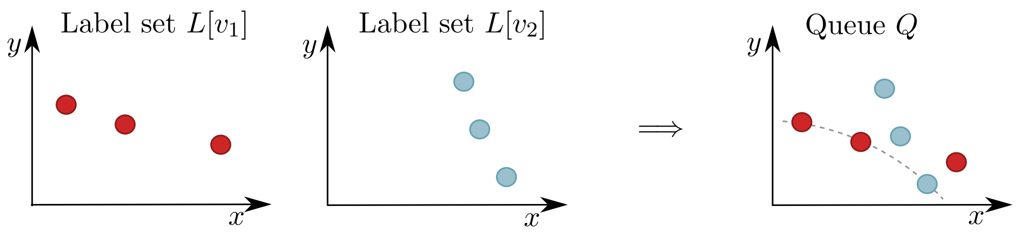


Figure 2.2.: Locally Pareto optimal labels within their label set L and within the queue of all tentative labels Q: Locally optimal labels might not be globally optimal, i.e., not all labels in Q are part of the (dotted) Pareto front.

Algorithm 2.1 presents the basic structure of label setting and correcting approaches: In each iteration of its outer loop, the algorithm retrieves a label l from Q. Let this label belong to a node u. The algorithm proceeds to derive a candidate label l' for each outgoing edge (u, v). The candidate label l' is discarded if it is dominated by any label in the label set $L[v]$. Otherwise, it is added to $L[v]$ and Q. Labels of v that are dominated by l' are removed. The outer loop of the algorithm finishes once all tentative labels in Q have been explored without leading to new non-dominated candidate labels. The resulting labels in the label sets of nodes can now be backtracked to derive the Pareto optimal paths (not shown here).

The queue Q contains tentative labels of different nodes. While labels in a label set of a single node do not dominate each other, labels within Q may dominate each other. Figure 2.2 visualizes this for a fast plain queue populated with labels of two different nodes.

Label setting algorithms remove only non-dominated labels from Q. These labels are not only locally optimal within their respective label sets, but also globally optimal in Q with respect to any other label to be explored next. They cannot be dominated by any other label discovered later, as their costs are only increasing and never decreasing.[1] Label setting is therefore optimal in the sense that all extracted labels are part of the output of Pareto optimal paths [60], i.e., extracted labels are permanent.

Bi-objective label setting has first been proposed by Hansen [27] and was later generalized to the multi-objective case by Martins [40]. Q has commonly been implemented as a lexicographically sorted priority queue, as relying on Observation 2.1, its minimal element is known to be Pareto optimal. Current research on label setting algorithms focuses on different label selection strategies [29, 47], different kinds of priority queue implementations [47, 53], speedup techniques applicable for one-to-one (single target) searches (e.g., [17, 53]), and parallelism due to Sanders and Mandow [60].

Label correcting algorithms are less sophisticated in their label choice and just select *any* tentative label for the next iteration by using a simple queue data structure (*FIFO*, *LIFO*, ...). Suboptimal labels may be extracted and processed, which do not belong to any Pareto optimal path. Only once the queue has run completely empty, the label sets can no longer change (i.e., tentative labels become permanent). Bi-criteria label correcting algorithms have been subject to extensive research on selection strategies (e.g., node selection instead of label selection), implementation choices and speedup techniques just like label setting algorithms [65, 11, 47, 63].

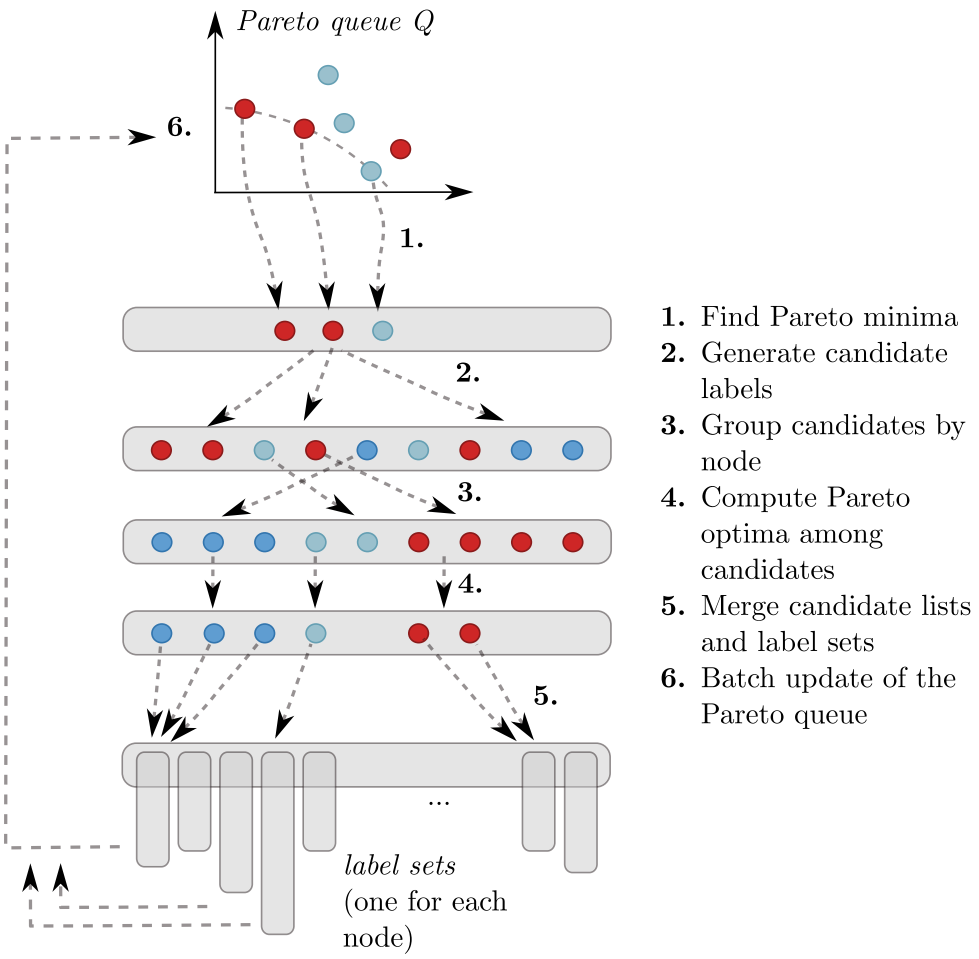
The choice between label setting and label correcting algorithms comes with a trade-off between time spend in queue operations managing the extraction of (Pareto optimal) labels and time spend processing suboptimal labels. Computational results indicate that no method is clearly superior [23, 47, 54]. Garroppo et al. [21] support these results with their theoretical analysis of label setting and label correcting approaches.

[1]For a detailed proof see [60].

2.3. Parallel Pareto Search

Sanders and Mandow [60] present a parallel label setting algorithm called `paPaSearch`, short for *parallel Pareto Search*. While classic label setting algorithms select *any* Pareto optimal label from Q per iteration, `paPaSearch` extracts *all* of them.[2] It relies on a multi-dimensional generalization of a priority queue, the *Pareto queue*, to find and extract these independent labels. The algorithm then processes them in parallel.

We outline the bi-criteria version of the algorithm, before detailing label sets and the Pareto queue. Figure 2.3 presents the algorithm. A detailed description follows.



Pareto queue Q
6.
1.
2.
3.
4.
5.
...
label sets (one for each node)
1. Find Pareto minima
2. Generate candidate labels
3. Group candidates by node
4. Compute Pareto optima among candidates
5. Merge candidate lists and label sets
6. Batch update of the Pareto queue

Figure 2.3.: Outline of the `paPaSearch` algorithm of Sanders and Mandow [60], consisting of six steps, which are repeated in a loop until the queue runs empty. Labels are represented as dots, with their color encoding the corresponding node.

[2]Not to be confused with label correcting algorithms, which extract any tentative label.

1. **Find Pareto minima:** Extract all Pareto optimal labels from the Pareto queue.
2. **Generate candidate labels:** For each Pareto minima and each outgoing edge of the corresponding node generate a candidate label.
3. **Group candidates:** To prepare the batch processing of labels, group candidate labels by the node they are assigned to (e.g., via sorting).
4. **Compute Pareto optima among candidates:** Independently for each node, drop candidate labels dominated by other candidate labels for the same node. If the candidate labels are sorted lexicographically (e.g., due to appropriate sorting in the previous step), Observation 2.2 details the correspondence of Pareto optima to labels with prefix minimal y-coordinates. The latter can be identified in a single scan.
5. **Merge candidate lists and label sets:** Independently for each node, merge surviving candidate labels with the label sets of the respective node. Candidates dominated by an old label or equal to an old label are dropped. All other candidates are inserted and may lead to several tentative labels being dominated and deleted. All modifications of the label set are logged in order to apply the same updates to Q.
6. **Batch update of the Pareto queue:** Update Q by applying all updates gathered in the previous step. The Pareto queue now contains all remaining and new tentative labels of which a Pareto optimal subset can be extracted for the next iteration.

The individual steps of the algorithm (including all data structure operations) are meant to be performed by several threads in parallel, i.e., each thread extracts a subset of labels from Q, creates a subset of candidate labels, merges a subset of labels into label sets, etc. Sanders and Mandow [60] specify their algorithm using parallel primitives such as task scheduling with randomized work stealing and prefix sums for load balancing. For example in steps 2-5, prefix sums are used to assign equal numbers of candidate labels to each thread. This implies that labels belonging to the same node may be distributed over multiple threads, requiring their collaboration.

Sanders and Mandow [60] detail a bi-criteria Pareto queue and bi-criteria label sets, both which are meant to be implemented as parallel trees with bulk updates.

Pareto Queue.

The Pareto queue is a simplified version of the data structure described in [10] (and in some papers cited there). It stores tentative labels lexicographically sorted in the leaves of a balanced binary search tree.

In contrast to a simple priority queue, it allows to extract *all* Pareto optimal labels available in the current iteration, instead of just one of them. As stated in Observation 2.2, Pareto minima are prefix minima over the y-coordinates of lexicographically sorted sequences. While prefix minima can easily be found sequentially by scanning all leaves from left to right, the serial dependencies on the y-coordinates of predecessors need to be broken for a parallel computation: Each inner node v stores a pointer to the left-most leaf with the minimum y value in the subtree rooted at v, i.e., the prefix minima imposed by its subtree. When searching a subtree for Pareto minima, we only descend into subtrees whose minimal label is not dominated by the minimal label of a preceding subtree. Procedure 2.2 shows the pseudo code of this computation. The recursive calls in lines 8 and 9 are independent; both subtrees can be searched in parallel using task parallelism (see Definition 2.5).

For an implementation of the parallel batch update of the Pareto queue at the end of an iteration, Sanders and Mandow [60] refer to the parallel red-black tree of Frias and Singler [19] (which we review in Section 3.1).

Procedure 2.2: findParetoMinima(v, u)

```
Input: Binary tree node $v$, prefix minima $u$ imposed by a preceding subtree
Output: all Pareto optimal labels in the tree rooted in $v$
1 if $v$ is leaf then
2     report the label stored at $v$
3 else if $v \to left.min$ dominates $v \to right.min$ then
4     findParetoMinima($v \to left$, u)
5 else if $u$ dominates $v \to left.min$ then
6     findParetoMinima($v \to right$, u)
7 else in parallel
8     findParetoMinima($v \to left$, u)                    // Only report values not
9     findParetoMinima($v \to right$, $v \to left.min$)   // dominated by the left subtree
```

Label Sets.

Similar to classic sequential labeling algorithms, labels assigned to nodes are stored in label sets. However, as a main difference, dominance checks and updates have to consider sequences of several candidate labels instead of just individual ones.

The Pareto optima in label sets are stored lexicographically ordered in the leaves of balanced search trees. According to Observation 2.3, such sets can be simultaneously sorted by increasing x-coordinates and by decreasing y-coordinates, enabling us to check dominance in logarithmic time in the size of the label set (i.e., the height of the tree):

Definition 2.8 (Predecessor Dominance Check)**.** Search the insertion position of a candidate label within the lexicographically sorted set. The candidate label is dominated if it is dominated by its predecessor label (the x-predecessor). If non-dominated, the candidate label may dominate other labels, i.e., all direct successor labels whose y-coordinate is larger or equal to its own. A second search therefore locates the first label with a y-coordinate smaller than the one of the candidate label (the y-predecessor). All labels between the x-predecessor and y-predecessor have to be deleted as they are dominated. A visual representation of this concept is given in Figure 2.4, highlighting the candidate label and its predecessors.

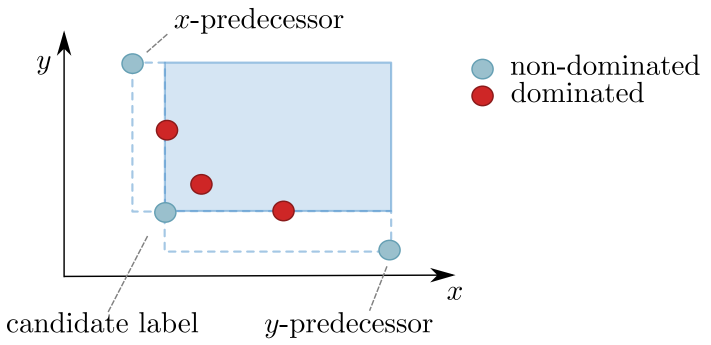


Figure 2.4.: Insertion of a candidate label into a label set. As indicated by the shaded region, all labels between the x- and y-predecessors are dominated.

Once all dominance checks are completed, the label set can be updated in a second pass, removing all dominated labels, and inserting the surviving candidate labels in a single batch update. This batch update can again be realized using the approach of Frias and Singler [19].

Theoretical Results.

Sanders and Mandow [60] prove several qualities of their `paPaSearch` algorithm, which we just summarize here:

- As Dijkstra's [18] algorithm, it performs at most n iterations, where n is the number of nodes. This means all additional work of the multi-criteria case compared to the single criteria is completely parallelizable.
- It performs asymptotically less work than classic sequential label setting algorithms since the batched nature of data structure updates supplies additional information, enabling efficient update algorithms.

With our description in Chapter 4, we are the first to present an implementation of this algorithm.

3. Parallel Bulk Updates for Balanced Search Trees

The parallel algorithm of Sanders and Mandow [60] relies on batched data structure updates, i.e., sequences of insertions and deletions meant to update either the Pareto queue or a label set. The authors propose to implement both these data structures as balanced binary search trees with parallel bulk updates, such as the parallel red-black tree of Frias and Singler [19].

For an algorithm or data structure to be efficient in practice, it has to consider the memory hierarchies of modern hardware, including the block-wise transfer between individual cache levels. Motivated by this insight, we adopt a cache-efficient balanced tree data structure and augment it with parallel bulk updates, instead of using the proposed binary trees. We begin with a short review of the relevant literature before detailing our own approach. A thorough evaluation then follows in Section 3.3.

3.1. Related Work

Cache-efficient trees take various forms, with *cache-aware* [56, 13] and *cache-oblivious* B-trees [6, 7] being among the most prominent. B-trees are balanced trees that increase spatial locality by storing multiple elements per node. This makes them well suited for systems only efficient at transferring larger blocks of data. Accordingly, a simple cache-sensitive B-tree has a node size of a single cache line [55] and can be multiple times faster than binary trees [14]. Cache-sensitive memory layouts help to improve the performance of the latter but are not sufficient to outperform B-trees [57].

Reducing the height of a B-tree can further improve its performance, as the shorter search paths lead to fewer cache faults. The height of a B-tree can be reduced by increasing the branching factor of inner nodes. Rao and Ross [56] propose the *CSB$^+$-tree* that achieves this by storing all children of a node as a single consecutive memory block. A single pointer to this block is then sufficient to address all children, freeing up space to store more keys per cache-line. A complementary technique is proposed by Chen et al. [13]. Their *pB$^+$-tree* still maintains one pointer per child but increases the branching factor by making inner nodes larger (e.g., 8 cache lines wide). The latencies involved in the retrieval of these additional cache lines are then hidden with the help of prefetching. Hankins and Patel [26] furthermore show that larger nodes offer a good balance of cache misses, TLB[1] misses and instruction count. Further techniques improving the cache performance of B-tree are summarized by Graefe and Larson [22].

An I/O-optimal parallel tree data structure for the *Parallel External Memory (PEM)* model [4], the *parallel buffer tree*, is presented by Sitchinava and Zeh [62]. It is an adaptation of the sequential buffer tree [1], which delays updates directed to a subtree until enough operations have been gathered to make their processing worthwhile. The lazy execution of the buffer tree realizes a form of *bulk updates. Bulk updates* exploit temporal and spatial locality of otherwise individual consecutive updates by applying them in one operation. Such algorithms generally work by collecting and grouping updates directed to the same leaf into balanced update trees. These trees are then inserted into the original tree as a whole, followed by an appropriate rebalancing of the latter. If search paths of different keys have common prefixes, these only need to be traversed once [38]. Furthermore, rebalancing can be efficient since the whole update sequence can be considered instead of just an individual update [57]. Sequential bulk updates for *(a,b)*-trees, which include B-trees, have been studied in [51, 33]. An overview of bulk update algorithms for these and other kind of trees can be found in [57].

As Sitchinava and Zeh [62] point out, *concurrent data structures* allow multiple threads to operate simultaneously and independently on the same data. In *parallel data structures*, on the other hand, threads cooperate in order to speed up queries and updates. Concurrent bulk updates of B-trees have been considered by the database community (e.g., [51, 52, 34]) and depend on locks in order to synchronize threads. Recent results indicate performance advantages of lock-free (i.e., parallel) approaches [61].

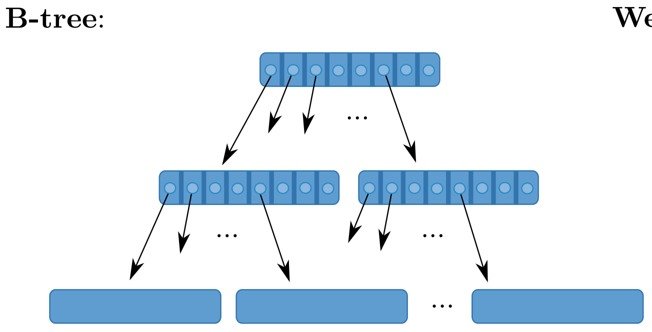
Practical work on parallel bulk updates is due to Frias and Singler [19]. The authors present a parallel bulk construction and insertion algorithm for *red-black trees* [24]. Their major tools are *split* and *concatenate* operations [67]: Given a sequence of elements to insert, the tree is splitted into p subtrees according to an initial fair distribution of the update sequence among the p threads. These subtrees can then be updated in parallel before they are concatenated to the result tree. To accommodate for different subtree sizes, the update procedure is augmented with load balancing based on randomized work stealing. The authors validate the practicality of their technique by reporting good speedups.

The bulk construction of Frias and Singler [19] shares ideas with a more theoretical algorithm proposed by Park and Park [48] for the *Parallel Random Access Machine (PRAM)*. Various other parallel search trees [12, 50, 28, 49] have been proposed for the *PRAM* model, all of which assume random access and rely on fine grained parallelism (e.g., pipelining or synchronized round-based computations).

3.2. Parallel Weight-balanced B-trees

We adopt B$^+$-trees [16] in order to augment them with parallel bulk updates. B$^+$-trees are a form of B-trees in which all elements reside in leaves. Internal nodes only store pointers to child nodes and router keys, serving as an index that guides search traversals to the correct leaves. Specifically, we implemented our tree as a *weight-balanced B-tree* [3], as it can be rebalanced efficiently (details follow). To adapt such a tree to the cache hierarchy of multicore machines, we follow the results of Hankins and Patel [26] by making internal nodes and leave nodes several cache lines wide.

[1]The translation look-aside buffer (TLB) caches recent translations from virtual to physical memory addresses.



Weight:	**Level**:
$w(v) \in \left[\frac{1}{4}\beta^2\alpha, \beta^2\alpha\right]$ $\in [1024, 4096]$	2
$w(v) \in \left[\frac{1}{4}\beta^1\alpha, \beta^1\alpha\right]$ $\in [128, 512]$	1
$w(v) \in \left[\frac{1}{4}\beta^0\alpha, \beta^0\alpha\right]$ $\in [16, 64]$	0

Figure 3.1.: Structure of a weight-balanced B-tree for the parameters $\alpha = 64$ and $\beta = 8$.

3.2.1. Weight-balanced B-trees

Weight-balanced B-trees were introduced by Arge and Vitter [3]. Arge [2] provides the following definitions.

Definition 3.1 (Weight)**.** The *weight* $w(v)$ of a leaf v in a search tree $\mathcal{T}$ is defined as the number of elements stored in it. The weight of an internal node v is defined as the sum of the weights of all its children (i.e., the number of elements in the leaves of the subtree rooted in v).

Definition 3.2 (Weight-balanced B-tree)**.** $\mathcal{T}$ is a *weight-balanced* B-tree with leaf parameter α and branching parameter β ($\alpha, \beta \geq 8$) if the following conditions hold:

- A leaf u has a weight $\frac{1}{4}a \leq w(u) \leq \alpha$. All leaves of $\mathcal{T}$ are on the same level (level 0).
- An internal node v on level l has weight $w(v) \leq \beta^l\alpha$.
- Except for the root, an internal node v on level l has weight $w(v) \geq \frac{1}{4}\beta^l\alpha$.
- The root has more than one child.

Figure 3.1 gives a concrete example for this abstract definition using the parameters $\alpha = 64$ and $\beta = 8$, showing the minimal and maximal acceptable subtree weight on each level of the tree. In addition to the presented child pointers and router keys within an inner node, an implementation has to store the total weight for each subtree.

Observation 3.1 (Weight-balanced B-Tree Properties)**.** *Weight-balanced B-trees* have a few important properties [3, 2]:

- The height of a N element weight-balanced B-tree with parameters α and β is $\mathcal{O}(\log_\beta(N/\alpha))$.
- The weight constraints imply a bounded degree: Any node v (except for the root) has between $\frac{1}{4}\beta$ and 4β weight-balanced children. A degree out of these bounds is not possible without violating the weight constraints of v.
- A node v only needs to be rebalanced every $\Theta(w(v))$ operations applied to its subtree.

Particularly the last property makes them interesting for us (summarizing the description in [2]): We can use a simple amortized argument to charge the costs of rebalancing a node v to all $\Theta(w(v))$ operations responsible for its imbalance. After inserting an element in a leaf, several nodes on the path from the leaf to the root may be out of balance. We can either apply regular B-tree rebalancing operations on these nodes or make use of *partial-rebuilding* (e.g., see [46]) and rebuild the tree rooted in the highest unbalanced node. As a subtree can be rebuild in a linear number of steps, this gives us an $\mathcal{O}(\log N)$ amortized update bound.

We proceed to explain our own contribution based on the existing ideas presented here.

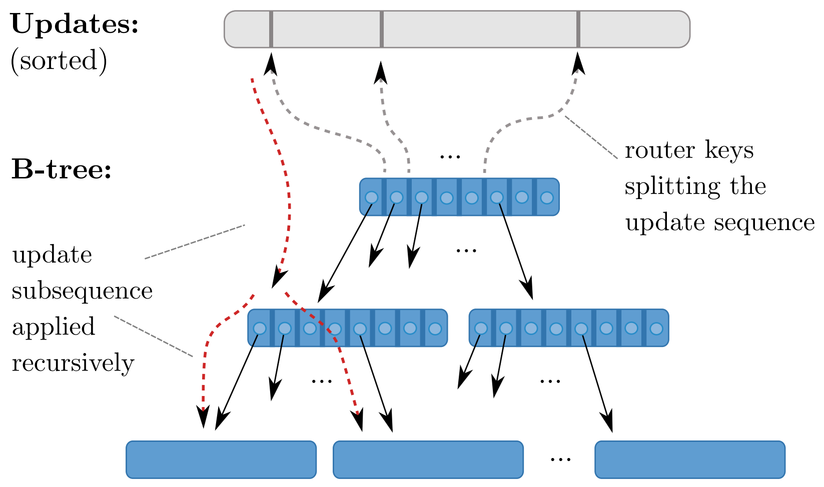

Updates:
(sorted)
B-tree:
router keys splitting the update sequence
update subsequence applied recursively
...

Figure 3.2.: A parallel bulk update procedure for B-trees using recursive tasks.

3.2.2. Parallel Bulk Updates

Given a sorted sequence of insertions and deletions, we want to update a tree in a single bulk operation using p collaborating threads. Figure 3.2 explains our general approach: We partition the update sequence according to the tree, i.e., we perform a binary search to locate the router keys of the root node within the sorted sequence and split it accordingly. We spawn a task (see Definition 2.5) for each resulting subsequence in order to apply it to its respective subtree. This is repeated recursively until all updates have been propagated to leaves. To update a leaf, we merge its elements with the incoming updates.

Splitting the update sequence according to the tree may lead to load imbalances if many updates are applied to the same area of the tree. However, this also means a subtree on *some* level needs to be rebalanced, as we can show by contradiction: Assuming we apply *many* updates in the same area of the tree without having to perform any rebalancing, then these updates have to be distributed over several leaves, as a single leaf can only store a constant number of elements. Updating several leaves is parallelized by our task concept and thus not a load balancing problem. We have therefore shown that load imbalances can only occur if a subtree needs to be rebalanced due to the over- or underflow of leaves. Fortunately, as explained next, we can parallelize this rebalancing.

3.2.3. Parallel Partial-Rebuilding

Let N be the size of the tree, k the length of a sequence of updates S, and p the number of threads involved in the computation. Let S_i denote the i'th update in S and let S_{ij} denote a range of updates. Furthermore, let d_{ij} be the *weight-delta* of an update range S_{ij}, i.e., the number of insertions minus the number of deletions in this range. The weight-delta d_{ij} realized by an update range S_{ij} is given as $d_{ij} = W_j - W_i$, where W is a sequence which we precompute using a single prefix sum before applying a bulk update to our tree. The first entry in W is initialized with zero. An entry W_i is set to $W_{i-1} + 1$ if S_{i-1} is an insertion or to $W_{i-1} - 1$ if it is a deletion.

The key insight of our approach is that we know whether a tree can be updated in-place or whether it needs to be rebalanced prior to descending into it: Let t be a subtree on level l. Furthermore, let S_{ij} be the range of updates to be applied to t. If the sum of the corresponding weight-delta d_{ij} and the original weight of the subtree do no longer fall into the acceptable weight range for a subtree on level l, then t needs to be rebalanced instead of just updated.

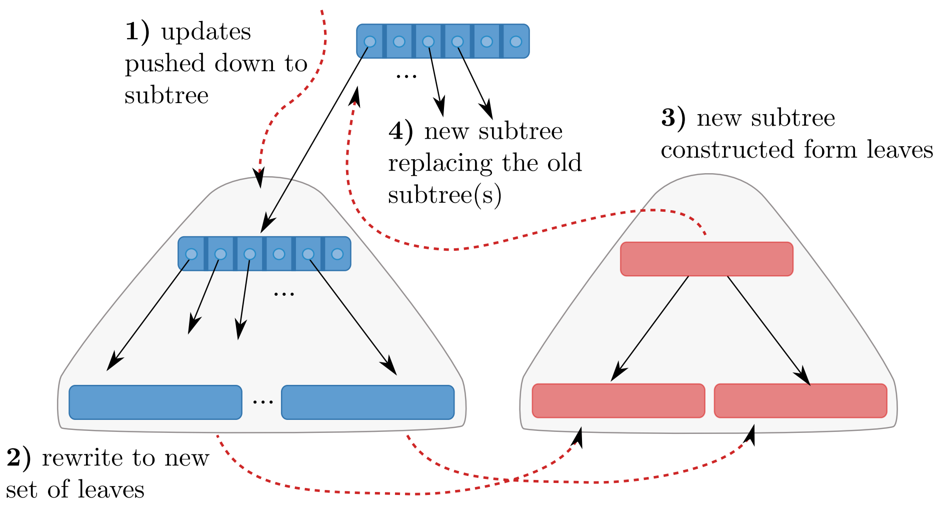

1) updates pushed down to subtree
...
4) new subtree replacing the old subtree(s)
3) new subtree constructed form leaves
...
...
2) rewrite to new set of leaves

Figure 3.3.: Parallel rebalancing of a weight-balanced B-tree using *partial-rebuilding*. If a subtree requires rebalancing it is rebuild from scratch from the existing elements and the incoming updates.

As stated in Section 3.2.2, a tree is updated by spawning recursive update tasks for its subtrees. To rebalance a tree, we use a very similar task parallel procedure. Figure 3.3 explains the idea: Updates are pushed down the tree using recursive tasks, but instead of merely updating leaves in place, the merged stream of elements and updates is written into a new set of leaves. Once all leaves are filled, a balanced tree is constructed from these leaves, replacing the old subtree. This means, rebalancing is not a separate step, but we can rebalance a tree while updating it.

The details of the parallel rebalancing approach depicted in Figure 3.3 follow. We note that tasks for individual subtrees are independent from each other. All synchronization that is required is already implicitly provided by the life cycle of tasks.

1. **Rebalancing Start:** We scan an inner node from left to right and compute the update ranges S_{ij} by using binary searches to locate the router keys within S. We use the weight-delta information to check which subtrees needs to be rebalanced and then trigger rebalancing tasks for each consecutive range[2] of unbalanced subtrees. In addition, we allocate an appropriate number of leaves to for all elements.

2. **Rewrite Process:** Within our recursive rebalancing tasks, we descend into all leaves of a subtree to be rebalanced. Elements in leaves are merged with the incoming updates and written to the new set of leaves. As an only exception, there is no need to descend into subtrees in which all elements are deleted, as indicated by weight-deltas leading to updated weights of zero.

   The position of an element within the new set of leaves (i.e., its updated rank) can be computed independently for each subtree using subtree weights and weight-deltas: Given a rank of zero for the root of the unbalanced subtree, the rank of a subtrees is given by the rank of its parent plus the updated total weight of all preceding siblings in the same node (as this is the space required for rewriting them). Following this approach, we know where to start writing the elements of a leaf. As ranks can be computed independently for different subtrees, the entire rewrite process can be parallelized using task parallelism.

[2]If a subtree or even a range of adjacent unbalanced subtrees has fewer elements (i.e., its total weight) than the size of a perfectly balanced tree for this level, we include the elements of an additional neighboring subtree.

Many updates may fall into a single leaf. We therefore split an update range applied to a leaf into subranges in order to process them in parallel. Let S_{ij} be an update range and let S_{pq} be one of its subranges. A binary search locates its first update in the existing keys of the leaf. The weight-delta information then helps to compute the starting position of elements resulting from the application of this particular update subrange S_{pq}: Let l be the index of the first element in the leaf smaller or equal to S_p. Given the rank r of the first element of the rewritten leaf, the rank of the first element produced by the application of the S_{pq} update range is $r + l + W_{ip}$.

3. **Tree Construction:** We want to build a perfectly balanced tree over the filled leaves. A perfectly balanced tree can take an equal number of insertions or deletions before it needs to be rebalanced, i.e., it is *half-full*. Accordingly, a weight-balanced B-tree on level l with leaf parameter α and branching parameter β has a designated weight of $\frac{1}{2}(\mathit{minweight}(l, \alpha, \beta) + \mathit{maxweight}(l, \alpha, \beta)) = \frac{1}{2}(\frac{1}{4}\beta^l\alpha + \beta^l\alpha) = \frac{5}{8}\beta^l\alpha$. For details, see Definition 3.2.

   With this size information at hand, a tree can be constructed using a top-down task parallel process: We recursively spawn tasks according to the layout of the balanced tree to be constructed that allocate and initialize the latter. When reaching the leaf level, instead of allocating leaves, the correctly pre-filled result leaves created in the previous step of this algorithm can be referenced.

4. **Replacement:** The tree construction yields an inner node with one or more children. These children have to be integrated into the parent node. Fortunately, as the degree of a node has well defined bounds (see Observation 3.1), there is always enough space within a node for all its children.

3.2.4. Runtime Analysis

Let N be the size of a weight-balanced B-tree, k size of the applied bulk update sequence, and p be the number of threads involved in the computation. To eliminate lower-order terms we assume $N \geq p^2$ and $k \geq p$. For simplicity of the following analysis, we furthermore treat the leaf parameter α and branching parameter β as constants. The parameters become significant again when we want to discuss the performance of the B-tree on actual hardware. We consider this problem in our evaluation in Section 3.3.1.

According to Definition 3.2 a sequential weight-balanced B-tree has a height of $\mathcal{O}(\log N)$ and an amortized update bound of $\mathcal{O}(\log N)$ for individual updates. As in [19], we note that the cost of a sequential bulk update is bound by the cost of applying the updates individually one by one, i.e., $\mathcal{O}(k \log N)$.

Our parallel weight-balanced B-tree is updated using tasks. Task parallelism with randomized work stealing performed by p threads leads to an execution time of $\mathcal{O}(T_1/p + T_\infty)$, where T_1 is the serial execution time of an operation and T_∞ the critical path of the computation, which cannot be parallelized [8]. If a bulk update of size k does *not* lead to an overflow of a tree of size N, then T_∞ is bound by its height $\mathcal{O}(\log N)$. In combination with the upper bound of the sequential bulk update time $T_1 = \mathcal{O}(k \log N)$, this gives us an amortized update bound of $\mathcal{O}(k/p \cdot \log N + \log N)$ for the parallel bulk update. We can simplify this bound to $\mathcal{O}(k/p \cdot \log N)$.

We also have to consider the case where the updates lead to an overflow of the tree and therefore to its reconstruction. This reconstructed tree has at most $N + k$ elements. Using p threads, its level-by-level construction takes $\mathcal{O}((N + k)/p)$ time due to the geometrically decreasing work per level. As indicated in Observation 3.1, we can charge these rebalancing costs to all $\Theta(N + k)$ updates leading to its imbalance. The update bound of $\mathcal{O}(k/p \cdot \log N)$ therefore stays valid even for this special case.

In addition to the actual update, our algorithm relies on a single parallel prefix sum as a pre-computation step. It runs in $\mathcal{O}(k/p + \log p)$ time (e.g., [32]) and is therefore dominated by the update time of $\mathcal{O}(k/p \cdot \log N)$. The same update bound is also achieved by Frias and Singler [19].

3.2.5. Implementation Details

We implemented our parallel B-tree in C++ using Intel® Threading Building Blocks (Intel® TBB 4.1)[3]. We shortly summarize a few details which positively affected the performance of our B-tree implementation:

- The B-tree supports bulk updates consisting of insertions, deletions and a mixture of both. The parallel prefix sum computation preceding a bulk update is however only required if a bulk contains insertions *and* deletions. Otherwise, weight-deltas are implicitly provided by the number of updates in an update range. Given this simple shortcut, we allow clients to specify the content of bulk updates (i.e., `insertions_only`, `deletions_only`, `mixed`) to disable the weight-delta computation if possible.
- Spawning a task for every subtree to be updated constitutes overhead if the work is already sufficiently balanced between the different threads. We therefore adopt a dynamic cutoff value used by the implementation of the parallel red-black tree of Frias and Singler [19]: A subtree is updated sequentially without spawning additional tasks if the number of updates in the corresponding update range is smaller than $(k/p)/\log(k/p)$, where k is the size of the entire bulk update and p the number of involved threads.
- To update a leaf, we sequentially merge the incoming updates with the existing elements of the leaf. The result is written to a thread-local spare leaf, which replaces the old leaf in the tree. The latter then becomes the spare leaf for the next update.
- Nodes are padded to cache line boundaries to prevent false sharing. They are allocated using the TBB scalable memory allocator [31].
- Rewriting a subtree requires a set of new result leaves. Our experiments indicated that is faster to allocate leaves lazily by the fist thread writing to a specific one, instead of allocating all leaves before the rewriting starts (even if done in parallel). The required synchronization can be realized with little overhead using an atomic `compare_and_swap` operation per allocated leaf.

3.3. Evaluation

We evaluate our B-tree on a system with two Octa-Core Intel Xeon E5-2670 CPUs (*Sandy Bridge*) clocked at 2.6 GHz. The sockets form two NUMA nodes, both maintaining 32 GB of main memory. Each of the eight cores per socket has a 64 KB private L1-Cache and a 256 KB private L2-Cache. In addition, all cores of a socket share a 20 MB L3-Cache. Our implementation is compiled with `-O3 -march=native -std=c++11` using GCC 4.7.2.

Our tests have the following properties (mostly adopted from [19]):

- **Data:** Elements are chosen to be similar to node-label pairs used by the Pareto queue of our parallel label setting algorithm, i.e., structs with three random 32-bit integers: x-coordinate, y-coordinate, and node id.

[3] `http://threadingbuildingblocks.org`

- **Skew:** Random values for the tree are taken from the entire integer value range $[0, 2^{32}]$. Random integers for updates are taken from the range $[0, skew \cdot 2^{32}]$, where $skew$ is a factor controlling the locality of insertions and deletions. As in [19], we consider uniformly distributed elements ($skew = 1$) and more localized updates that require rebalancing of the tree and load balancing among threads ($skew = 0.01$).
- **Ratio:** Bulk updates with k elements are applied to trees of size $N = r \cdot k$, where r is a configurable tree size ratio. Bulk construction of a tree is given for $r = 0$. As in [19], we only consider $r = 0$, $r = 0.1$ and $r = 10$.

  To generate a deletion batch, we perform a random shuffle of all N elements and select the first k elements. In order to accommodate for the skew factor, we only include elements of the tree in the shuffle whose value is smaller than $skew \cdot 2^{32}$.

We flush the data cache after the initial tree has been constructed and before the bulk sequence is generated and applied. All reported timing values are pruned averages[4] of the elapsed wall-clock time of otherwise unloaded machine, i.e., the average after the 25% fastest and slowest runtimes have been filtered. Each test is repeated at least[5] 50 times.

Furthermore, we have implemented a direct sequential counter part of our parallel B-tree that follows the same implementation strategy, but does not rely on parallel constructs such as tasks or parallel loops. All speedups are reported in regard to this sequential implementation.

3.3.1. Parameter Tuning

Before running the experiments, we have to determine the size of leaves and inner nodes by tuning the leaf parameter α and the branching parameter β.

For smaller nodes our B-tree behaves more like a binary tree, whereas for larger nodes it becomes more array-like. Both have their advantages and disadvantages, depending on the structure of the input sequences, i.e., the distance of update positions within the tree. Larger distances make smaller leaves promising due to the small number of accessed elements per applied update. Larger leaves, on the other hand, are advantageous for smaller distances, as multiple updates can be applied by scanning a leave (i.e., high spatial locality). Figure 3.4 visualizes this trade-off for the leaf parameter α: We are inserting a fixed set of uniformly distributed elements into trees of different sizes. As expected, extreme parameter settings are only good for extreme input sequences.

In order to identify specific parameter values for α and β, we have to run a detailed analysis. We inspect the uniform insertion of k elements into trees of size $N = 10 \cdot k$ and $N = 100 \cdot k$. The first ratio represents a dense update sequence, whereas the second one represents are sparse update sequence. Figures 3.5 and 3.6 show the results of these computations.

The TBB scalable memory allocator used by our B-tree implementation redirects uncached memory allocations over 8 KByte to the operating system, bypassing all thread local heaps [31]. We observed that this limits the scalability of our B-tree in cases where allocations are required (i.e., construction and rebalancing) and therefore only consider node sizes below this threshold.

According to Figure 3.5 and the limitations of the memory allocator, we set the parameters for dense updates to $\alpha = 660$ and $\beta = 32$. Furthermore, according to Figure 3.6, we set the parameters to $\alpha = 64$ and $\beta = 32$ for sparse updates.

[4]Bader et al. [5] propose to use such an approach if running times are not at least two or three orders of magnitude faster than the clock resolution.

[5]Smaller instances are repeated significantly more often, so that each test takes at least several seconds in total.

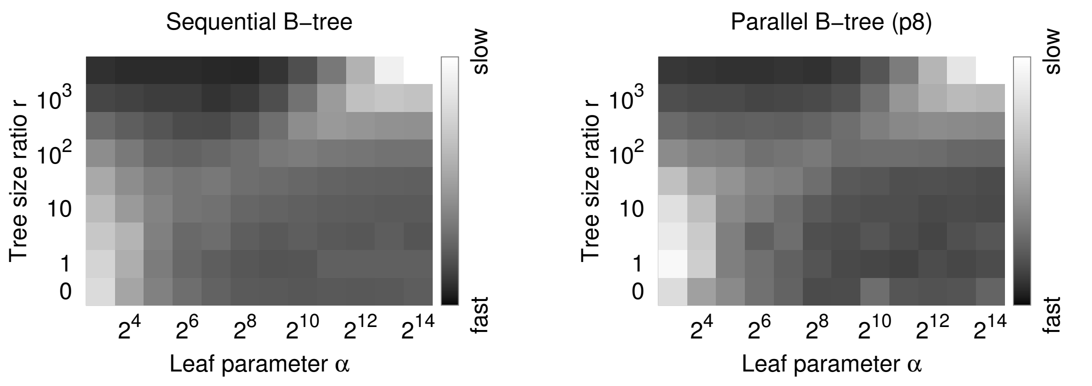


Figure 3.4.: Performance heatmap: Insertion of $k = 100\,000$ uniform random elements into a tree of size $N = r \cdot k$ for a varying leaf parameter α and varying tree sizes (scaled via r). The map is best read row-wise. The grayscale assigned to a leaf parameter indicates the relative performance of this parameter to the performance of all other leaf parameters for the same tree ratio. The results validate the hypothesis of small leaves being suitable for sparse insertions and large leaves being appropriate for dense insertions.

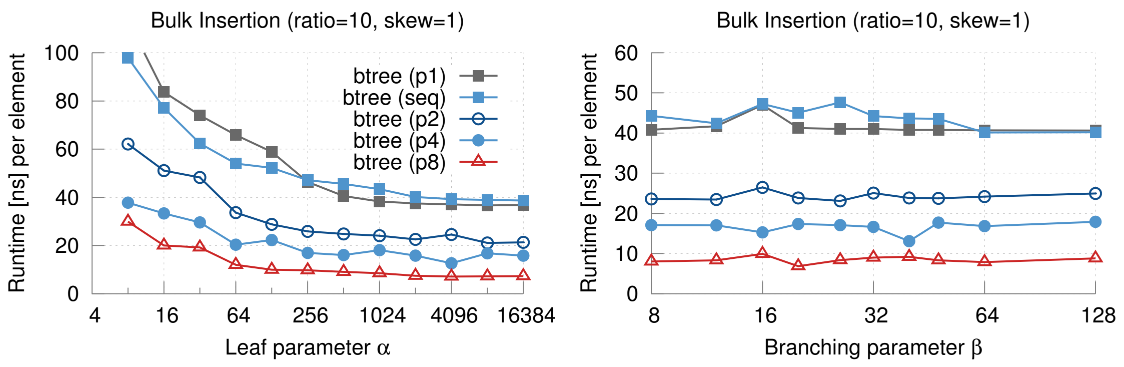


Figure 3.5.: Detailed parameter tuning for dense insertions: Insertion of $k = 100\,000$ elements into trees of size $N = 10 \cdot k$. If not varied in the experiment, the parameters are set to $\alpha = 660$ and $\beta = 32$. The left plot indicates performance advantages of larger leaves, whereas the right one shows that the size of inner nodes only has a moderate performance influence.

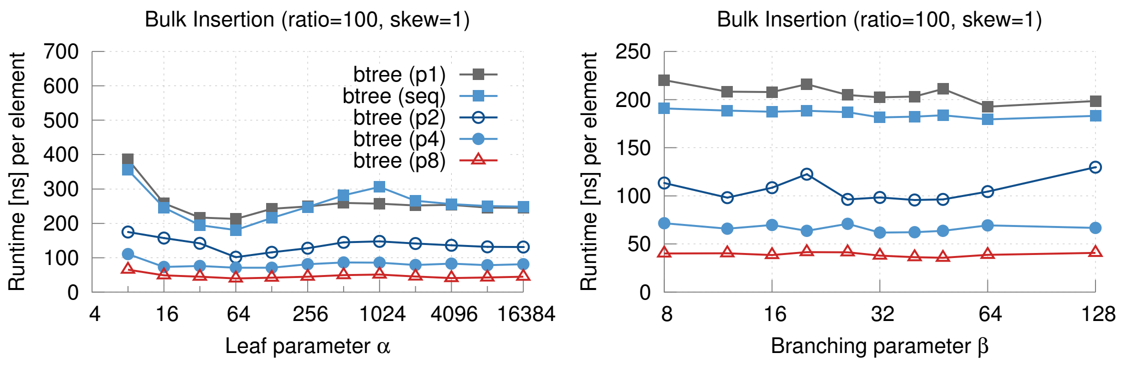


Figure 3.6.: Detailed parameter tuning for sparse insertions: Insertion of $k = 10\,000$ elements into trees of size $N = 100 \cdot k$ (thus the same tree size as in Figure 3.5). If not varied in the experiment, the parameters are set to $\alpha = 64$ and $\beta = 32$. The experiment renders a leaf parameter of about 64 as optimal. In addition, similar to Figure 3.5, the branching factor of nodes has only moderate influence on the performance of the tree.

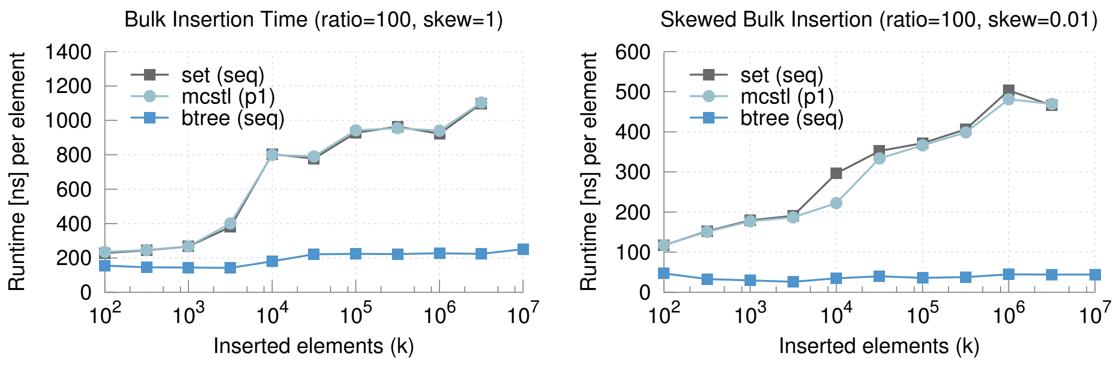


Figure 3.7.: Insertion of a sequence of elements into a tree 100-times larger than the sequence. The largest binary tree experiments do not fit into the main memory and are therefore omitted. Our B-tree outperforms both sequential competitors. They will therefore not be considered any further in the following experiments.

3.3.2. Experimental Results

We are not aware of any direct competitors besides the parallel red-black tree of Frias and Singler [19], an extension of the C++ `std::set` with parallel bulk updates. We adopt their latest implementation (MCSTL[6] 0.8.0-beta) and compile it with the latest compatible GCC version 4.2.4 using the flags `-O3 -march=native`. Within the following experiments, `mcstl (p1)` denotes the serial[7] execution of the bulk-update procedure and `mcstl (p8)` denotes the `MCSTL` red-black tree using eight threads.

In Figure 3.7 we use a simple experiment to gauge the performance of our sequential B-tree implementation compared to the performance of the MCSTL algorithm and an ordinary `std::set` binary tree without bulk updates. Our B-tree outperforms both single-threaded competitors. They are therefore not considered any further in the following experiments.

Figures 3.8, 3.11, 3.12, and 3.13 show the insertion into trees of different sizes, comparing our B-tree to the red-black tree of Frias and Singler [19] with their own benchmarking methodology. We complete these results by reporting our achieved absolute speedups. Figures 3.9 and 3.10 furthermore present deletions from a tree, a feature not supported by the parallel red-black tree. A summary of the results follows.

Performance.

Our B-tree is significantly faster than the red-black tree of Frias and Singler [19], outperforming it in all experiments. In most cases, even our sequential implementation is faster than their parallel red-black tree using eight threads. Exceptions to this last statement are very sparse insertions (Figure 3.13) where loading large leaves constitutes overhead if only very few element of a leave are affected by updates. In contrast to the binary tree, our B-tree furthermore shows a relatively constant insertion cost per element.

We attribute the good performance of our B-tree to its cache-efficient design. As an additional factor, according to Frias and Singler [19], node allocations constitute a considerable share of the work to be done. As B-tree nodes are larger, fewer nodes have to be allocated to store the same number of elements, mitigating the allocation problem.

[6]Multi-Core Standard Template Library, `http://algo2.iti.kit.edu/singler/mcstl/`

[7]A complete sequential implementation of the bulk update procedure does not exist. We therefore run the parallel version with just one thread.

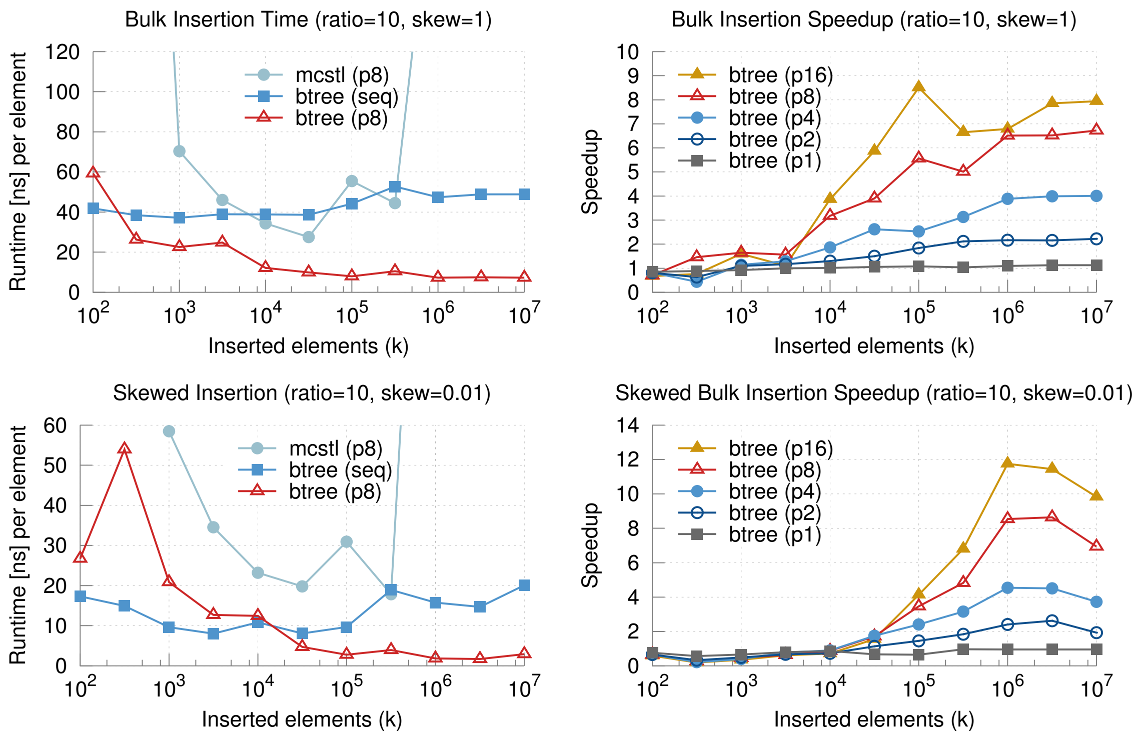


Figure 3.8.: Dense bulk insertion: Insertion of a sequence of elements into a tree that is 10-times larger than the sequence.

Speedup.

For uniform updates that do not require rebalancing ($skew = 1$) we achieve a speedup of three for four threads and six for eight threads (Figures 3.8, 3.9, 3.13). For the bulk construction experiment or experiments where rebalancing is required (Figures 3.8, 3.11, and 3.12), our maximal achieved speedups are higher and sometimes even slightly *super-linear*. We ascribe the superlinear speedup to the utilization fast private of caches which remain unused in the sequential case. For example, during the construction, tree nodes are allocated and connected to each other in a downward pass but only populated with data during the subsequent upward pass. The second pass is faster if large portions of the tree are still cached in higher levels of the cache hierarchy, enabling a multi-threaded version to reach super-linear speedups thanks to the additional caches.

Speedups are achieved only for a rather large number of elements. This result is consistent with the results of Frias and Singler [19] and can be improved by forcing a process with p threads to only use p cores of a *single* socket. The process will then profit from improved locality for small inputs (which is particularly important on a NUMA machine) but suffer from the limited I/O bandwidth of the single socket for larger ones. We followed Frias and Singler [19] and refrained from explicitly binding threads to specific sockets and cores.

Deletions.

Insertion and deletions are implemented almost identically and therefore have similar performance and speedups (see Figure 3.9). As an important difference however, weight-deltas may indicate the deletion of an *entire* subtree, relieving from having to push down updates to the leaves of this subtree. Instead, it can be deleted directly without significant computational effort. Such a deletion is very fast, making it difficult to achieve any speedups in a parallel implementation.

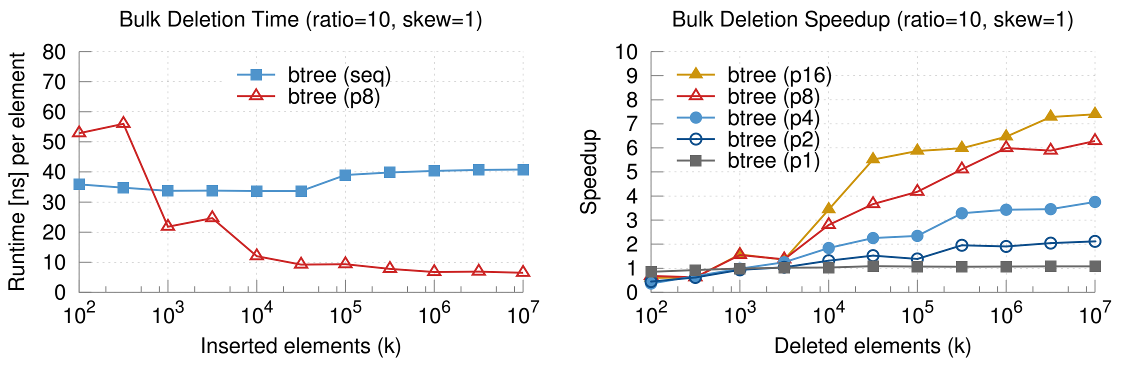


Figure 3.9.: Deletion of a sequence of size k from a tree of size $n = 10 \cdot k$.

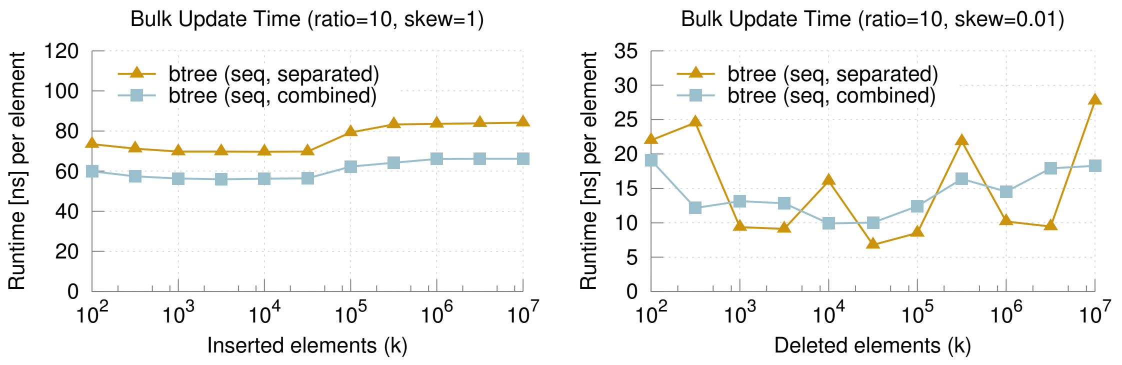


Figure 3.10.: Insertion and deletion: Update of trees which are 10-times larger than a given insertion and deletion sequence. *Separated* denotes the case where the insertion and deletion of k elements are applied to the tree as two separate bulk updates. *Combined* denotes the case where these two update sequences are merged and applied in a single bulk operation instead of two. Only the combined approach requires the prefix sum computation of the weight sequence (see Sections 3.2.3 and 3.2.5).

As explained in the implementation details presented in Section 3.2.5, the prefix sum computation can be enabled or disabled depending on the content of a bulk update. It is not required for update sequences containing either *only insertions* or *only deletions*. It was therefore disabled in all benchmarks performed so far.

Figure 3.10 details the case when we have to perform both deletions *and* insertions on the same tree. The result is two-fold: For uniformly distributed insertions and deletions ($skew = 1$) it is faster to update the tree in a single bulk update instead of a bulk deletion followed by a separate bulk insertion. This is true even with the overhead of the prefix sum computation required for the former. This result is somewhat expected, as the tree has to be traversed only once instead of twice. For skewed insertions ($skew = 0.01$) the result is less clear. As indicated in Figure 3.10, the performance depends on a combination of the deletion sequence and the tree layout. For specific constellations, separate deletions may be optimized by the deletion short-cut for entire subtrees described above. The separate bulk updates then turn out be faster than the combined case, because there is neither an expensive first pass (for the deletion) nor the additional prefix sum computation.

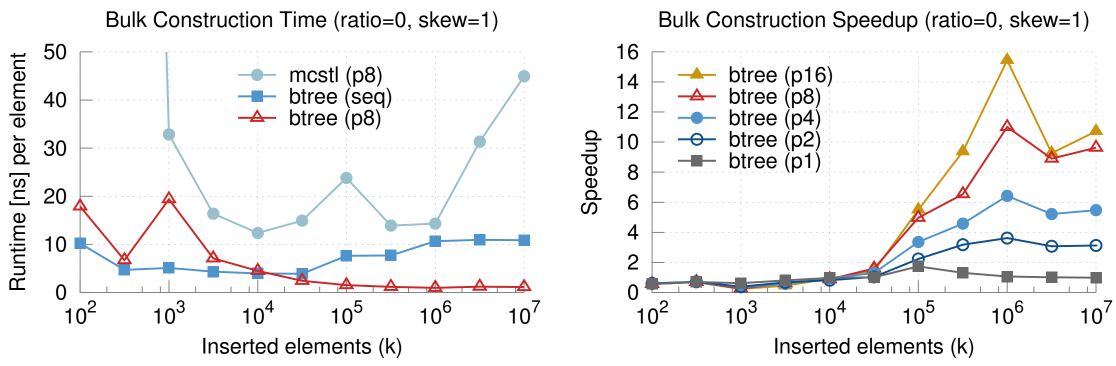


Figure 3.11.: Bulk construction by inserting a sequence of elements into an empty tree.

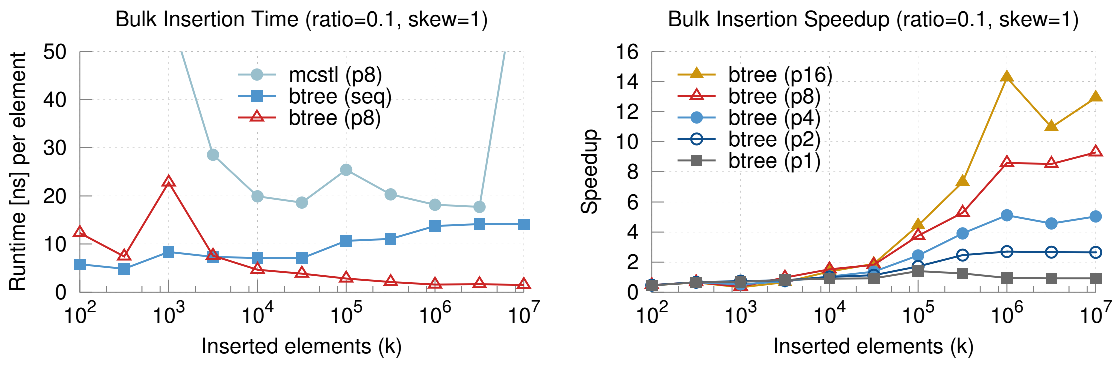


Figure 3.12.: Large bulk insertion: Insertion of a sequence that is 10-times larger than the tree, forcing a complete reconstruction of the latter.

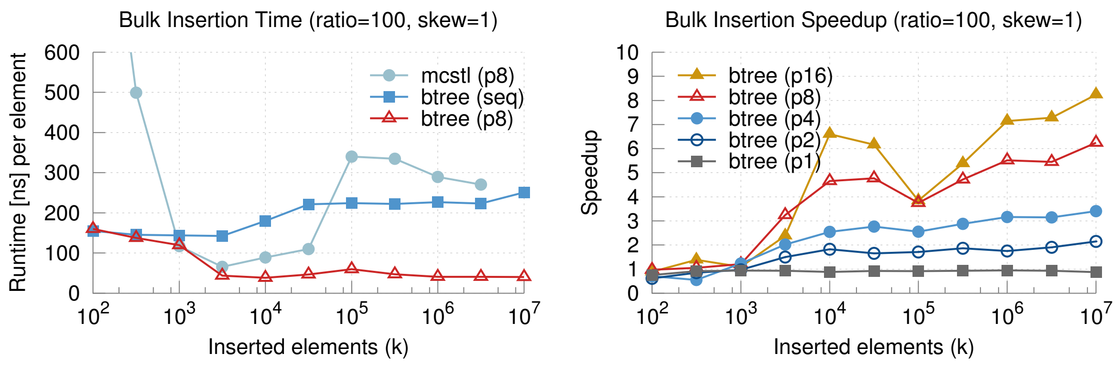


Figure 3.13.: Sparse bulk insertion: Insertion of a sequence of elements into a tree that is 100-times larger than the sequence. The largest experiment for the parallel red-black tree is missing because it does not fit into the main memory. Unfortunately, we have no proper explanation for the setback of the speedup of our B-tree for larger number of threads. We assume it is caused by a combination of a low branching factor in the root node and size of this tree compared to the size of the system caches, both making it difficult to reach an adequate performance if both sockets of the system have to be used.

4. Engineering the Parallel Pareto Search Algorithm

Goal of our engineering effort is to devise a fast and efficient implementation of the algorithm of Sanders and Mandow [60] for shared-memory multiprocessors. Consistent with the results in Chapter 3, we consider memory hierarchies, but also aim to reduce the need for excessive synchronization and load balancing. A detailed evaluation of our implementation follows in Chapter 5.

4.1. Coarse-grained Loop Parallelization

The parallel Pareto search algorithm explores several Pareto optimal paths in parallel. It extracts these paths from the Pareto queue in a main loop, until the queue runs empty. Even though *subsequent* loop iterations cannot be solved in parallel due to serial dependencies on the content of the queue, the work *within* an iteration can be parallelized.

As seen in Section 2.3, such an iteration consists of several steps, starting with an extraction of labels from the Pareto queue and ending with a batch update of the latter. Each of these steps is performed by several threads in parallel. Unfortunately, as Sanders and Mandow [60] point out that, such fine-grained parallelism may render the algorithm impractical. We approach this problem by adapting the algorithm slightly: Even though threads update different label sets in parallel, we update a single label set only sequentially by a single thread. Accordingly, to achieve proper load balancing, our implementation relies on the existence of a sufficiently large number of label sets to be updated per iteration. To decrease the number of synchronization and communication operations even further, we reduce the number of steps in the inner loop of the algorithm, as fewer steps imply fewer synchronization points.

Figure 4.1 outline the algorithm as implemented by us. A further description follows. Details on the involved data structures are then presented in the remainder of this chapter.

1. **Find Pareto minima:** Identify all Pareto optimal labels from the Pareto queue in parallel. For each encountered minima, generate candidate labels, and schedule the removal of the minima during in the later executed batch update of the Pareto queue Q.

   We detail this task parallel process in Section 4.3. It combines two steps of the original algorithm (*"Find Pareto minima"* and *"Generate candidate labels"*) and re-uses the load balancing performed for the Pareto minima computation to also balance the creation of candidates.

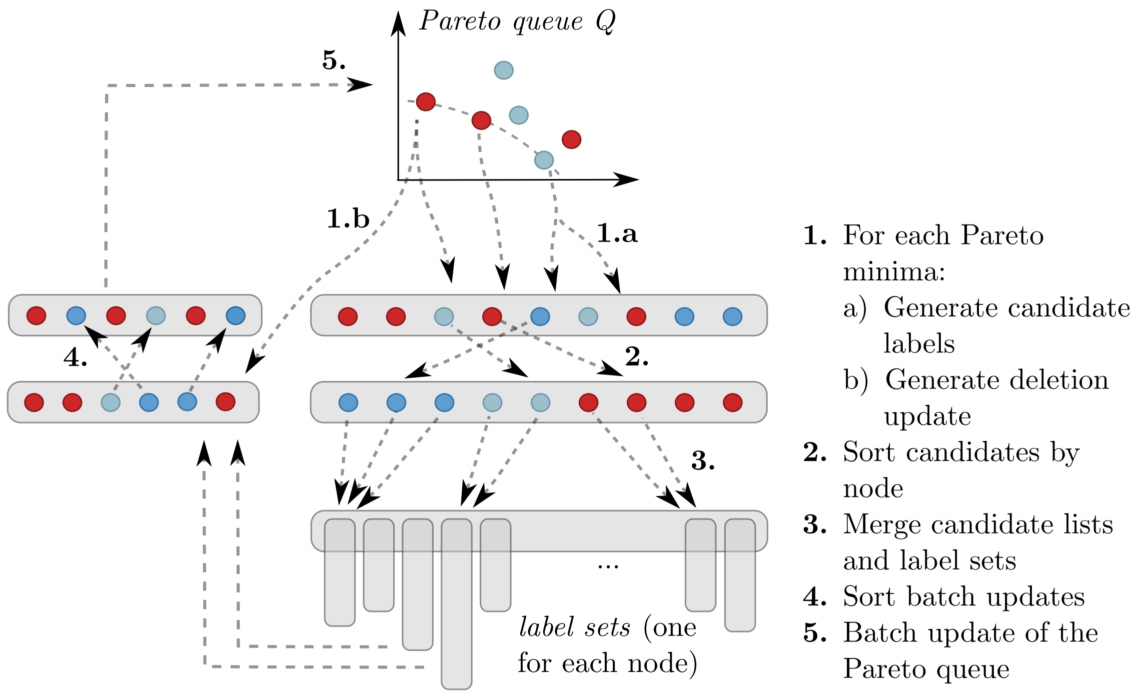

Pareto queue Q
5.
1.b
1.a
4.
2.
3.
...
label sets (one for each node)
1. For each Pareto minima:
a) Generate candidate labels
b) Generate deletion update
2. Sort candidates by node
3. Merge candidate lists and label sets
4. Sort batch updates
5. Batch update of the Pareto queue

Figure 4.1.: Outline of the shared-memory adaptation of the `paPaSearch` algorithm. Labels in their different forms are all represented as dots, with the color encoding the corresponding node.

2. **Sort candidates by node:** To prepare the batch processing of labels, use a parallel shared-memory sorting procedure to group candidate labels by the node they are assigned to.

   This implies all threads have written their generated candidate labels to a single, shared array. In Section 4.2, we present an approach of how such an array can be filled simultaneously with only moderate synchronization effort.

3. **Batch Update of label sets:** In parallel but independently for each node, merge candidate labels with the label sets of the respective node. All resulting modifications of the label set are furthermore collected as operations for the batch update of Q.

   Load balancing can be realized via a parallel loop, splitting the array of candidates into ranges of dynamic size depending on the remaining workload. Candidate labels belonging to the same node are identified using scanning. Each corresponding label set is updated sequentially by a single thread.

   This step combines the original steps *"Compute Pareto optima among candidates"* and *"Merge candidate lists and label sets"* into a single step. As explained in Section 4.4, this combination enables an optimized candidates filtering.

4. **Sort batch updates:** Sort all Pareto queue updates which have been gathered over the course of the current iteration lexicographically. Analogues to step two, this implies all these updates have been written to a shared array. Further details are provided in Section 4.2.

5. **Batch update of the Pareto queue:** Update Q by applying the updates that have been sorted in the previous step. For further details, see Section 4.3.

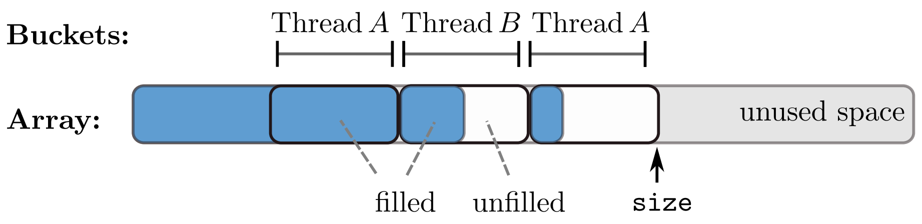


Figure 4.2.: Buckets within a shared array which are filled by different threads in parallel. Threads allocate a new bucket by atomically incrementing the `size` variable according to a fixed bucket size.

4.2. Collecting Results

The sorting steps of our algorithm adaptation (steps 2 and 4) require elements to be collected in contiguous arrays. These elements are generated by different threads in parallel. Unfortunately, the threads are likely to generate unequal number of elements. Due to the dynamic nature of the algorithm, even the total number of generated elements per iteration is unknown.

We implement a dynamic solution for this problem, enabling threads to append their locally generated results to a shared array. Even though this technique seems rather trivial, it is the key to our shared-memory approach, as it settles the important implementation detail of how to pass data from one step to the next.

Inspired by the multi-core radix sort of Wassenberg and Sanders [66], we allocate an array large enough to store all possible results. It may even be larger than the available main memory, because virtual memory implementations of modern operating systems map physical memory only on the first access of a memory page. We let all threads write to this array in parallel, but force them to synchronize their accesses using atomic operations on a shared size variable.

An atomic `fetch-and-add` of the size variable returns the value of the variable, and, in the same cycle, increments it with the given value. Let the returned value be x and let it be incremented by i. No other thread can interrupt this atomic increment. The range $[x, x+i)$ is therefore exclusively owned by the incrementing thread. As visualized in Figure 4.2, the range defines a bucket, which can be filled without further synchronization. Once this bucket is filled, the thread can request a new bucket by another atomic increment.

Threads always perform increments by a fixed bucket size. In contrast to increments for just one element, this reduces contention on the atomic, but also eliminates false sharing, as buckets can be aligned to cache line boundaries.

The last bucket of a thread can probably not be filled completely, resulting in gaps in the shared array without meaningful data. Fortunately, the next step of our algorithm is always to sort the collected data. By populating these gaps with a maximal value, we can use the sorting operation to move the corresponding entries to the end of the array, where they can be excluded from further processing. The approach comes with a trade-off between the frequency of atomic updates and the increased sort volume. We consider it in our evaluation in Section 5.3 by tuning the bucket size accordingly.

4.3. Implementing a Parallel Pareto Queue

The Pareto queue proposed by Sanders and Mandow [60] is based on a binary search tree (for details, see Section 2.3). As we have seen in Section 3.3.2, our parallel B-tree significantly outperforms its binary tree-based competitor. We therefore adopt it as the basis for our custom Pareto queue implementation.

Procedure 4.1: findParetoMinima(t, p, G)

```
Input: B-tree node $t$, prefix minima $p$ imposed by a preceding subtree, Graph
       $G = (V, E)$
Output: Candidate labels and deletion updates for all discovered Pareto optimal
        labels in the tree rooted in node $t$
$min \leftarrow p$                                             // current prefix minima
if $v$ is leaf then
    foreach node label pair $(u, l)$ in $t$ do                 // scan leaf for Pareto minima
        if $l_y < min_y$ or $l = min$ then                     // is $l$ is non-dominated?
            store deletion update for $l$
            foreach $(u, v) \in E$ do
                $w \leftarrow weight((u, v))$
                $l' \leftarrow (l_x + w_x, l_y + w_y)$
                store candidate label $(v, l')$
            $min \leftarrow l$
else
    foreach subtree slot $i$ in $t$ do                         // search subtrees in parallel
        if $t[i].min_y < min_y$ or $t[i].min = min$ then
            spawn task findParetoMinima($t[i].node$, $min$, $G$)
            $min \leftarrow t[i].min$
```

In a Pareto queue, the root of each subtree must be augmented with information about the label with the minimal y-coordinate in this subtree. Storing a pointer to this label, as proposed by Sanders and Mandow [60], would lead to additional (random) memory accesses during the discovery of Pareto minima. We therefore opt to increase the spatial locality by copying the x- and y-coordinates of these minimal labels to the corresponding internal nodes of our B-tree. As shown in our adapted pseudo code in Algorithm 4.1, internal nodes can then be scanned efficiently.

We use tasks to search different subtrees independently in parallel and rely on randomized work stealing for load balancing. We process each discovered Pareto minima immediately (see lines 5-9), i.e., we generate candidate labels and create a deletion update to remove the respective minima.

It is more efficient to update the Pareto queue in a single instead of multiple independent updates (for details, see Figure 3.10 and the description in Section 3.3.2)[1]. We therefore merge the Pareto minima deletions with the updates generated by label set modifications and apply them in a single bulk update. Our B-tree requires this update sequence to be sorted. In a sequential implementation, the Pareto minima can be extracted in lexicographic order, not requiring any further sorting. Therefore only the label set updates need to be sorted before both can be merged to form the complete update sequence. In the parallel implementation on the other hand, this merge shortcut does not apply, because the randomized work stealing of the task parallelism prevents an ordered extraction of minima.

[1]This applies at least to uniform updates. Fortunately, as shown in the appendix (see Figure A.5), this is the common usage pattern of the Pareto queue.

4.4. Implementing Label Set Updates

Given a sequence of candidate labels for a node, we have to check which of them are dominated and which need to be inserted into the label set. Labels of the label set which are dominated by newly inserted candidates have to be removed. We begin with a short description of the dominance check before detailing two alternative label set implementations.

In a preprocessing step, all candidate labels of a node are sorted lexicographically. Even though this could be achieved in the previous grouping step by using an appropriate comparison-based sort, a delayed lexicographic sort profits from simplified sort comparators (i.e., fewer branch) and improved cache locality. It furthermore enables us to overlap I/O and computation by prefetching a label set while the associated candidate labels are sorted.

When performing dominance checks, it is inefficient to spend work proportional to the number of old labels, which can be significantly larger than the number of candidates to insert. The predecessor-based dominance (see Definition 2.8) is therefore promising as it allows us to identify Pareto optimal candidate labels in logarithmic time in the size of the lexicographically ordered label set: A candidate label is dominated if it is dominated by its x-predecessor. A non-dominated candidate label furthermore dominates all labels between its x- and y-predecessors. The predecessor searches can either be realized as traversals in a search tree, or as binary searches on array-based label set implementations. We note that the predecessor approach also yields the correct insertion position of non-dominated labels.

Sanders and Mandow [60] propose to implement dominance checks for a set of candidate labels in two passes. In a first pass, candidates dominated by other candidates for the same node are meant to be filtered. The second pass shall then be used to compare the surviving candidates to the existing labels in the corresponding label set. In our implementation, we combine both steps in a pipelined fashion, reusing information gathered in the second step to improve the first one: We scan over candidate labels from left to right and check their dominance using the predecessor method referenced in the previous passage. Let l' be such a candidate and let l be the predecessor label dominating it. The candidates are sorted; all following candidates therefore have x-coordinates larger or equal to ${l'}_x$. Accordingly, following candidates are dominated if they have a y-coordinate larger or equal to l_y. We can filter them in constant time by a simple comparison to l_y without having to use the more costly logarithmic approach. Whereas the original proposal of filtering dominated labels among candidates amounted to only about 3%–5% of dominated labels being pre-filtered, our pipelined approach is able to filter 40–80%, depending on the problem instance and the label set implementation.

We present two label set implementations: A tree-based variant with logarithmic insertion time, and an array-based variant which is simpler but only offers linear insertion time.

Vector Label Set.

We store labels in lexicographically sorted unbounded arrays [41], i.e., in a `std::vector`. Such a vector provides fast random access and an automatic resize of the underlying array for insertions. Even though insertion and deletions are fast at the end, applying them to arbitrary positions requires linear time due to the shift of following elements.

In Figure 4.3, we present a label set update procedure attempting to combine insertions and deletions in order to minimize the number of shifted elements: Inserting a label may lead to existing labels being dominated. Instead of deleting these immediately, the range of dominated labels can be marked as a gap. This gap can then be filled by subsequently

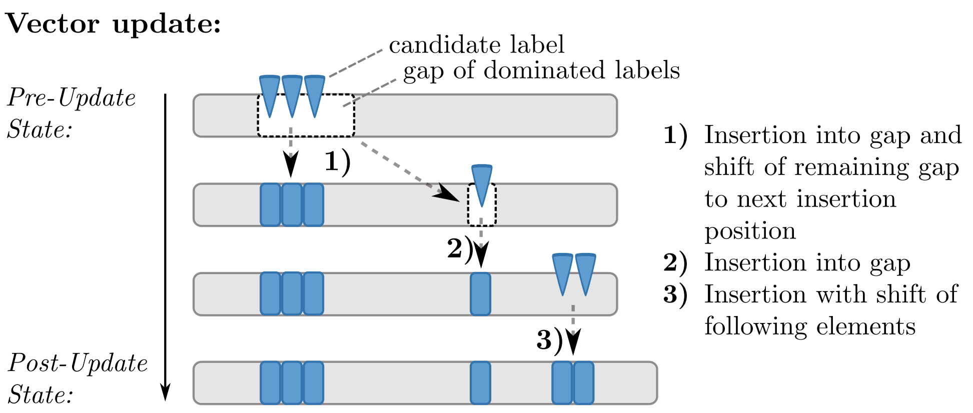

Vector update:
candidate label
gap of dominated labels
Pre-Update State:
Post-Update State:
1)
2)
3)
1) Insertion into gap and shift of remaining gap to next insertion position
2) Insertion into gap
3) Insertion with shift of following elements

Figure 4.3.: Example of a batch update of a vector-based label sets. The update process is visualized using four states, beginning with the pre-update state, and ending with the post-update state: In a single pass over the vector, six lexicographically sorted candidate labels are inserted. This leads to the domination four existing labels, which are removed by being replaced.

inserted labels without requiring additional shifts. If the next label to be inserted is not adjacent to the gap, the gap can be moved to the correct insertion position (i.e., Step 2 in Figure 4.3). Instead of shifting *all* following elements in the vector, therefore only the elements between the gap and the next label insertion position need to be shifted. Once all candidates in the current batch have been inserted, there may be a remaining gap. We delete it in a final operation, leaving the label set as a contiguous array of labels.

Following this approach, updating a label set requires only a *single pass* with neither an old label nor a candidate label being looked at twice. The worst-case runtime of inserting a batch of candidates into a label set is linear in the size of both. In practice however, insertions can be significantly faster, i.e., if insertions are concentrated at the end of the vector or if most shifts can be circumvented using the gaps created by dominated labels.

B-tree Label Set.

Following the results of Chapter 3, we implement the tree-based label set using a sequential version of our weight-balanced B-tree. It is updated in two passes. We check which labels are dominated in a first pass and apply the corresponding updates in a second one, by using its bulk update capabilities.

Similar to the B-tree update procedure (see Section 3.2.2), candidate labels are pushed down the tree to the lexicographically sorted leaves using recursive update procedures. Within leaves, dominance can be checked with an implementation of the predecessor dominance based on scanning (see Definition 2.8). However, it is not sufficient to search for dominated labels just within these leaves. Figure 4.4 presents the specific case of a candidate label dominating labels in an adjacent leaf. Fortunately, this implies that also the router key of the updated subtree is dominated. When we back out of the recursive update procedure of a subtree, we can therefore use the router key to check if a descend into its right sibling is required in order to identify the remaining dominated labels.[2]

As an additional use-case, router keys help to realize a further improved candidate filtering, which is also apparent in Figure 4.4: Candidate labels have to be propagated to a subtree only if they are not already dominated by the router key of the preceding subtree. Many

[2]To a certain extent, this resembles the *finger search* [25] idea which is used for updates in the more work efficient extension of the `paPaSearch` algorithm [60].

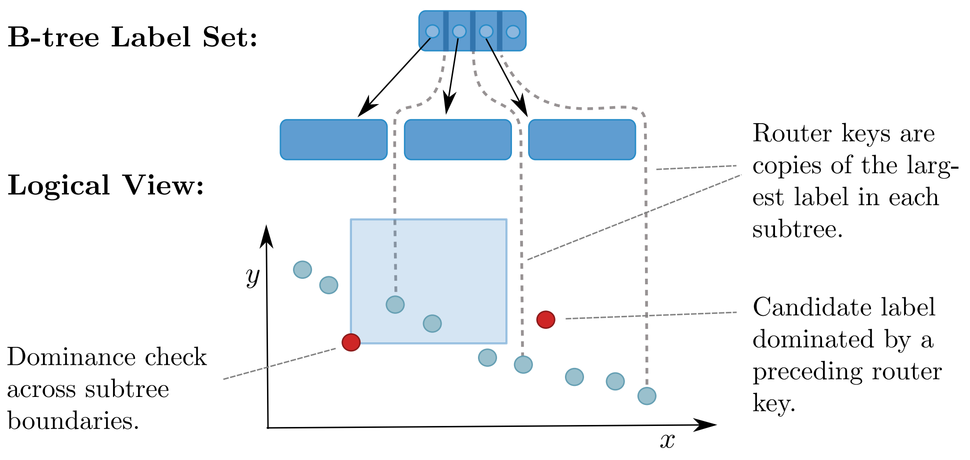

B-tree Label Set:
Logical View:
Router keys are copies of the largest label in each subtree.
Candidate label dominated by a preceding router key.
Dominance check across subtree boundaries.
y
x

Figure 4.4.: Structure of a B-tree-based label set. Router keys help to detect when dominance check span across subtrees. In addition, they can be used to filter candidate labels in inner nodes and without having to access the corresponding subtree.

filtering and dominance checks can therefore be carried out in internal nodes without having to propagate the candidates to leaves.

Major disadvantage of the approach, as presented so far, are its two passes over the tree. i.e., the first one for dominance checks and the according generation of updates and the second one to apply these updates. To optimize the second pass, we adopt a form of *relaxed balance* [45]: If the updates generated for a leaf keep the weight of this leaf between the minimal and maximal acceptable value (see Definition 3.2), we apply them immediately. This requires a second pass over the leaf but prevents another traversal of the root-to-leaf path. However, as learned in Section 3.2.1, updating a leaf may unbalance other nodes on this path. We therefore slightly *relax* the weight-balancing condition by deferring these rebalancing operations to the next proper bulk update of the affected subtree. Relaxed balancing maintains the current structure of the tree; only the number of elements per leaf can change. Nodes are only created or deleted during the proper bulk updates.

5. Evaluation

We have implemented our approach in C++ using Intel® Threading Building Blocks (Intel® TBB 4.1)[1]. All allocations are performed using the TBB scalable memory allocator [31]. We have furthermore implemented a direct sequential counterpart that follows the same implementation strategy, but does not rely on parallel constructs such as tasks or parallel loops. We compile our code it with GCC 4.7.2 and the flags `-O3 -march=native -std=c++11`.

We test our implementations on a system with two Octa-Core Intel Xeon E5-2670 CPUs (*Sandy Bridge*) clocked at 2.6 GHz. The sockets form two NUMA nodes, both maintaining 32 GB of main memory. Each of the eight cores per socket has a 64 KB private L1-Cache and a 256 KB private L2-Cache. In addition, all cores of a socket share a 20 MB L3-Cache. Each socket has four memory channels. The system runs Suse Linux Enterprise (SLES) 11 with kernel version 3.0.42.

5.1. Sequential Competitor

To serve as a competitor for the parallel label setting algorithm, we are in need of an equally well-tuned reference implementation of a sequential bi-criteria shortest path algorithm. We opt to implement a classic label setting algorithm, as it is the conceptually closest alternative. As detailed in Section 2.2, there are various key design decisions involved in such an implementation. We settle them one by one:

Queue Data Structure: We store all tentative labels in an addressable priority queue [41] based on a tuned binary heap similar to the one proposed in [58].

In the context of label setting algorithms, binary heaps have been shown to be highly competitive with other implementation choices such as Fibonacci heaps or bucket data structures [47, 53]. Binary heaps have also been chosen by Raith and Ehrgott [54].

We tried to reduce the size of this heap by storing only a single label per node, i.e., only the best label instead all of them. In this context, we also experimented with different strategies to find the next best label of a node once its best label was removed from the heap. However, none of these approaches turned out to be as competitive and consistent in their results as just storing all labels within a single heap.

[1] `http://threadingbuildingblocks.org/`

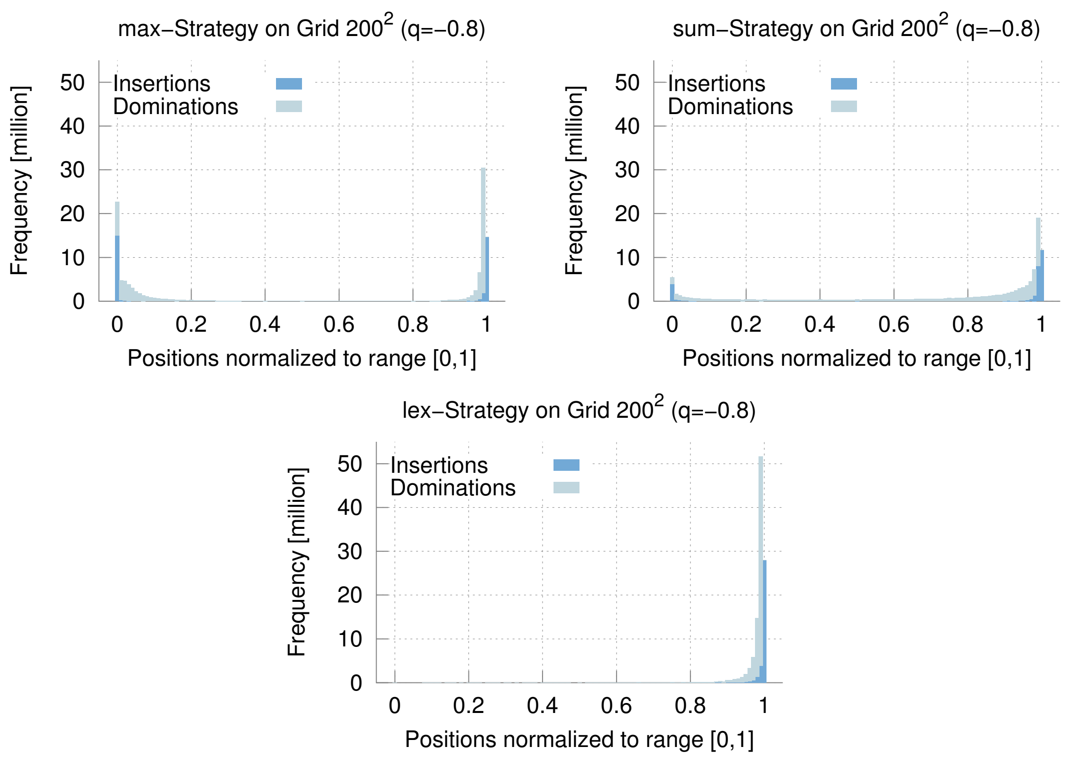


Figure 5.1.: Classic bi-criteria label setting: Label set operations for different label selection strategies: In contrast to `sum` and `max`, the `lex` label selection strategy leads to an access pattern that is almost exclusively concentrated at the end of a lexicographically ordered label set. Figure B.6 in the appendix shows that this behavior also holds for other instances.

Label Selection Strategy: Our priority queue is lexicographically sorted.

This design decision stands in contrast to recent experimental results [47, 29] on label selection strategies that identified alternative orderings to be more efficient, such as selecting the label with the minimal sum (i.e., $x + y$) or the label whose largest criteria (i.e., $max\{x, y\}$) is minimal.

We deviate from these results because a label selection strategy cannot be seen in isolation: The selection of different Pareto optimal labels from the priority queue leads to a different access pattern on label sets, and, depending on the dominance check, to a different number of label comparisons.

Figure 5.1 presents the label set access pattern for lexicographically sorted sets, i.e. the positions within a label set where candidate labels are either dominated or inserted. The lexicographic label selection strategy leads to an access pattern that is almost exclusively concentrated at the end of a label set. We proceed to show how this makes lexicographic label selection favorable for us.

Label Set Data Structure: We store labels in lexicographically sorted unbounded arrays [41].

Given that insertions happen almost exclusively at the end of our label sets, we can choose to implement them as *unbounded arrays* [41] (i.e., `std::vector`), which are very efficient at appending elements.

We have also experimented with search tree data structures (i.e., `std::set`) but these were clearly outperformed. Considering related work, we are not aware of any discussion on the best data structure choice for label sets.

Dominance Check: We check if a candidate label is dominated by the last label. If this is not the case, we fall back to an implementation of the predecessor method (see Definition 2.8).

The dominance short cut relies on the focused access pattern of the lexicographic ordering. As a fall back, we use an implementation of the predecessor method (for further details, see Definition 2.8) to check for dominance, and to compute the appropriate insertion position into the sorted set. The pseudo code of the resulting dominance check is presented in Algorithm 7.1. Our whole insertion procedure is then shown in Algorithm 7.2. Both algorithms are available in the appendix.

In Figure 5.2, we compare our classic label setting implementation (`LSetClassic`) to the label setting and label correcting implementations of Raith and Ehrgott [54].

The results of [54] have been gathered on a system with a 2.40 GHz Intel Core 2 Duo processor and 2 GB RAM. The algorithms are implemented in C and compiled with GCC 4.1.1 with compile option `-O3`. The results of our C++ implementation are gathered on a very similar machine (1.83 GHz Intel Core 2 Duo, 3 GB RAM, GCC 4.4). We therefore compare these implementations by directly comparing the reported timing values.

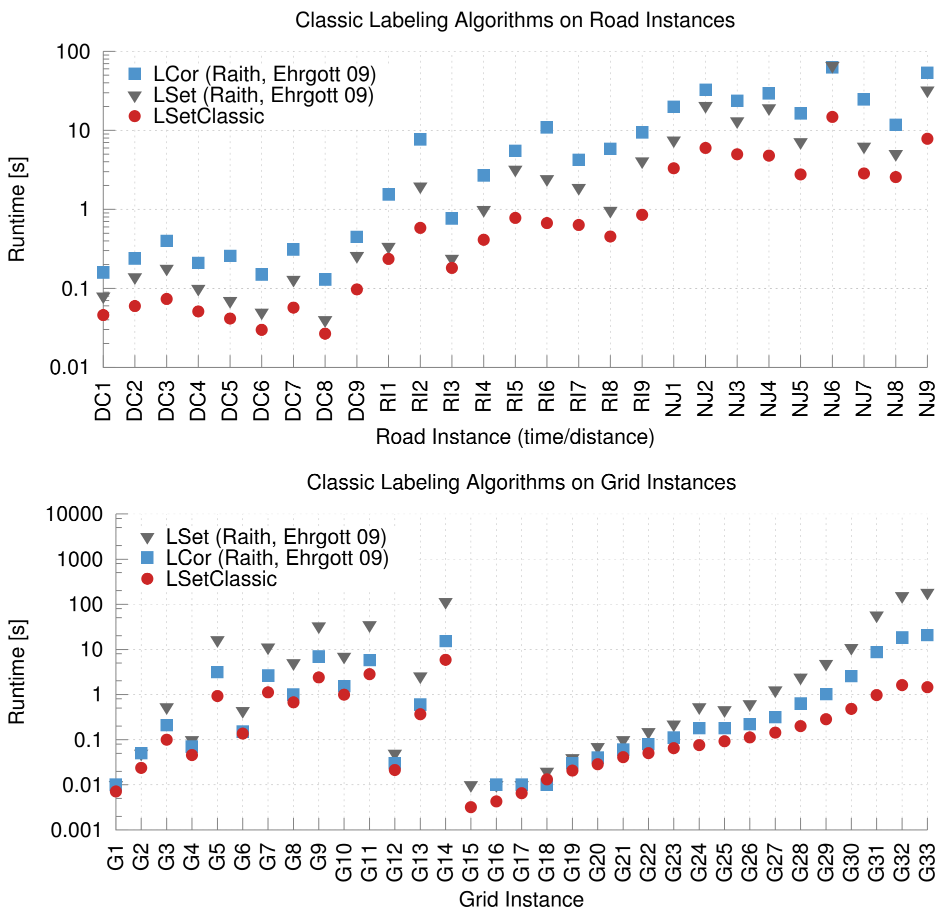


Figure 5.2.: Comparison of our classic label setting implementation (`LSetClassic`) to label setting and label correcting implementations of Raith and Ehrgott [54] on road and grid instances of the same authors. Details on the different road instances are provided in Table 5.1. A description of the grid instances can be found in the Appendix B.

Even given our slightly slower CPU, our implementations turns out to be faster for almost[2] all instances. We conclude that `LSetClassic` can therefore serve as a faithful reference for our `paPaSearch` implementation.

5.2. Test Instances

The crucial parameter affecting the performance of bi-criteria search algorithms is the total number of Pareto optima [44]. The latter is not only related to the size and shape of the graph, but also to the correlation between objectives [43, 11]. The number of Pareto optimal paths decreases with increasing correlations and even degenerates to the single-criterion case for perfect correlation.

We consider several different instances within this thesis. Most of these instances have been proposed in the context of directed search from a start node s to a target node t (i.e., *one-to-one* search). We adopt the concept of reporting the number of labels of the target node, but still compute the Pareto optimal paths to all other nodes (*one-to-all* search).

Random Grids.

Grids organize nodes into a rectangular shape of a given height and width. Each node is connected to its (at most) four neighboring nodes via directed edges with random edge weights.

Grids are very common in the literature (e.g., [54, 23, 47]). We adopt a definition that, in addition to the number of nodes, allows a controlled adjustment of the correlation between objectives [37, 43]. Let G be square grid and let $-1 \leq q \leq 1$ be a parameter controlling the correlation objectives. Each edge weight $\vec{w} = (x, y)$ in G is then computed as follows: x and y^* are independent random integers uniformly selected from the interval $[1, c_{max}]$ and y is derived from x and y^* according to the correlation q:

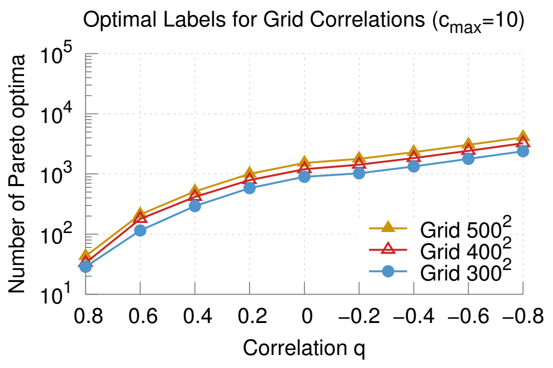
$$y = \begin{cases} q \cdot x + (1-q) \cdot y^* & \text{for} \quad q \geq 0 \\ 1 + c_{max} - (|q| \cdot x + (1-|q|) \cdot y^*) & \text{for} \quad q < 0 \end{cases}$$

Within our experiments, we consider positively correlated grids ($q = 0.8$), uncorrelated grids ($q = 0$) and negatively correlated grids ($q = -0.8$), both for $c_{max} = 10$ and $c_{max} = 1\,000$ (e.g., values used in [37, 47]). The characteristics of the different type of grid instances are detailed in Figure 5.3. Searches always start in one corner of the grid and end in the opposite corner.

Sensor Networks.

A sensor in a sensor network monitors its surrounding, processes gathered data, and communicates with its local peers. We consider artificially created sensor networks in which sensors are represented by nodes and communication partners connected via edges. As the two objectives, we chose the distance to the connected node and a random integer, both uncorrelated values form the range $[1, 10\,000]$. We use sensor networks with a fixed number of 100 000 nodes and node degree ranging from 5 to 50, i.e., degrees drastically different from the rather limited degree of the other instances.

[2]With the only exception of grid instance whose computation is faster than 0.01 seconds.



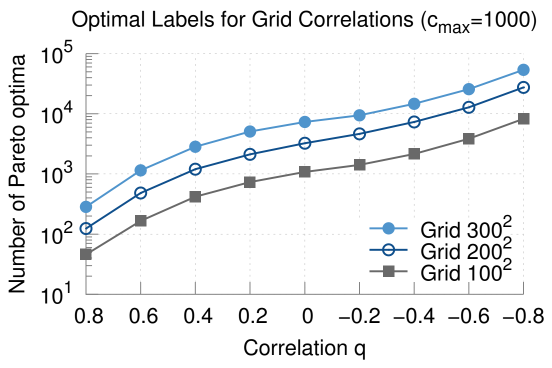


Figure 5.3.: Number of Pareto optimal labels for grid instances with different sizes and correlations: Larger graphs lead to more Pareto optimal paths. On the other hand, smaller maximal costs imply more identical labels and thus fewer Pareto optimal paths. The number of optima refers to the number of optimal paths from one corner of the grid to the opposite corner, not counting Pareto optimal paths to other nodes.

Name	*State*	*Nodes*	*Edges*	*Degree*	*Pareto optima*	
				Avg.	Avg.	Max
DC1-DC9	Washington, DC	9 559	39 377	4.12	3.33	7
RI1-RI9	Rhode Island	53 658	192 084	3.58	9.44	22
NJ1-NJ9	New Jersey	330 386	1 202 458	3.64	10.44	21
NY1-NY20	New York City	264 346	730 100	2.76	2 082.60	7 397

Table 5.1.: Characteristics of the used road instances. The number of Pareto optima refers to the number of optimal paths between pairs of random start and target nodes, not counting Pareto optimal paths to other nodes.

Road Networks.

We adopt road networks as real world use cases. We consider simple road instances with correlated objectives *(time/distance)* and more difficult road instances with uncorrelated objectives *(time/economic costs)*. Statistics on the road maps can be found in Table 5.1.

- *(time/distance):* Raith and Ehrgott [54] adopt the three US road maps as tests beds for bi-criteria shortest path search, i.e., Washington DC (DC), Rhode Island (RI) and New Jersey (NJ). They generate nine different problem instances for each map. The objectives time and distance are highly correlated (Pearson's correlation coefficiency of 0.99 [35]), leading to only few Pareto optimal paths.
- *(time/economic costs):* Machuca and Mandow [36] present a road instance of New York City (NY) featuring the objectives time and economic cost. The latter attempts to model the cost of traversing an edge, as a combination of estimated fuel costs and highway tools. The two objectives are uncorrelated (Pearson's correlation coefficiency of 0.16), leading to a moderately difficult problem instance.

5.3. Parameter and Implementation Choices

Based on the parameter tuning of the weight-balanced B-tree presented in Section 3.3.1, we configure the Pareto queue for dense updates ($\alpha = 660$, $\beta = 32$) and B-tree label set for rather sparse ones ($\alpha = 64$, $\beta = 32$).

The bucket append technique described in Section 4.2 depends on the bucket size parameter. This parameter realizes a trade-off between the frequency of threads synchronizations

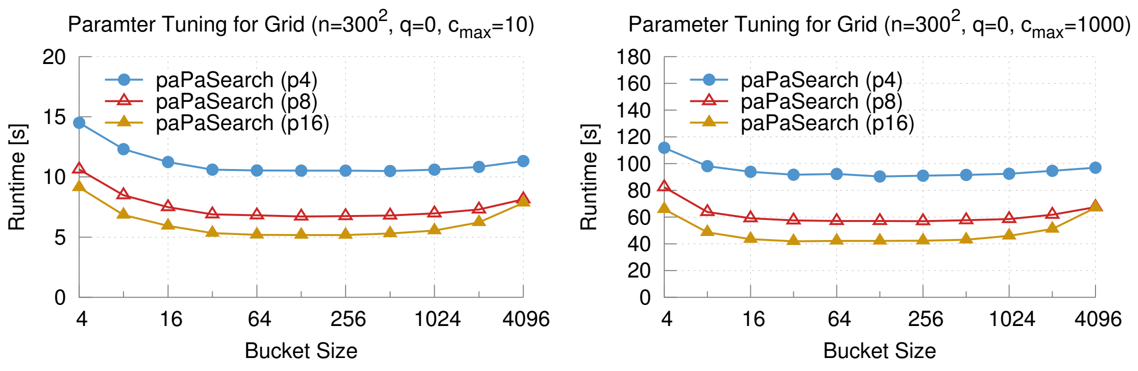


Figure 5.4.: Tuning of the bucket size parameter for uncorrelated grids of size $n = 300^2$. Only extreme parameter values have an impact on the performance, any other value is acceptable, e.g., a batch size of 128.

through atomic updates and an increased sort volume. According to the tuning results of Figure 5.4, we set the bucket size to 128.

In Section 4.4, we introduced one label set version based on our weight-balanced B-tree and one based on `std::vector`. Figure 5.5 compares these two label set implementations according to their performance on a grid configuration (other configurations yield similar results and are therefore omitted). The results can be traced back to the access pattern of the `paPaSearch` algorithm on label sets. Figure 5.6 presents this pattern. It supports the hypothesis that the vector cannot handle all access patterns equally well. It will therefore not be considered any further in this thesis.

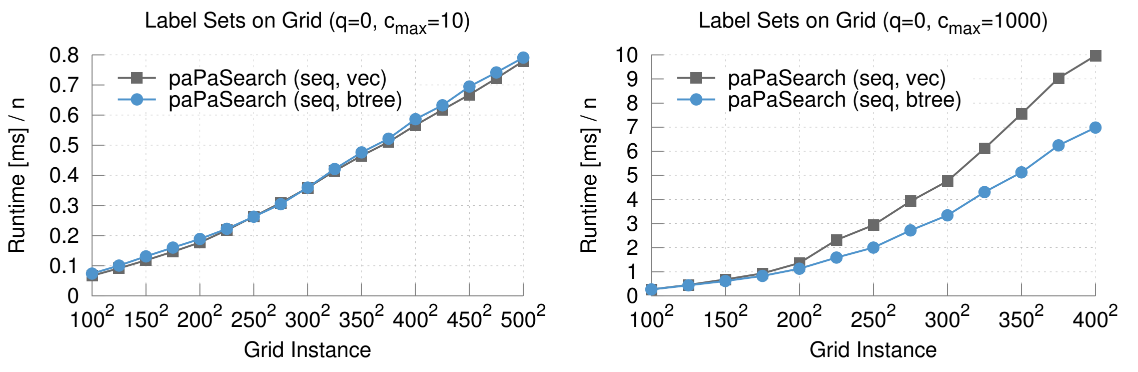


Figure 5.5.: Grid experiments for the two different label set implementations. The result indicates that the vector is not competitive for all instances.

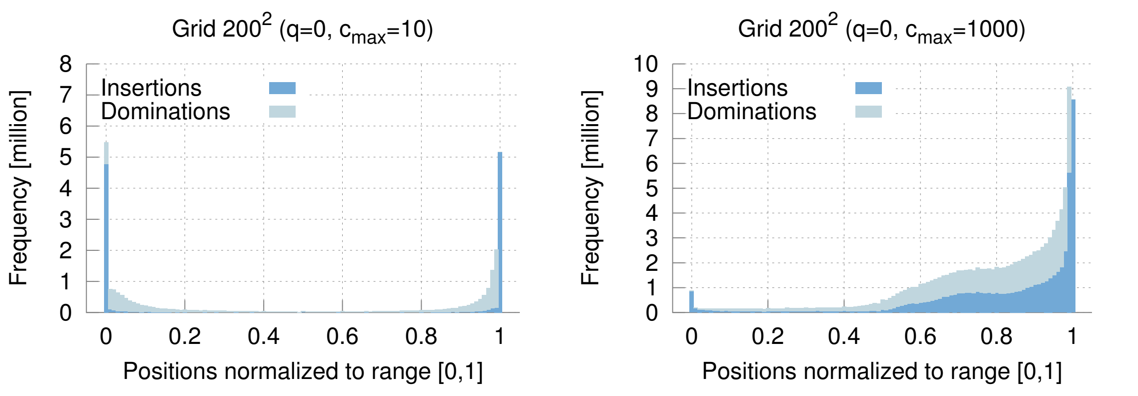


Figure 5.6.: Position within a label set where candidate labels are either dominated or inserted by our parallel labeling algorithm. A similar analysis for other access pattern can be found in the appendix (Figure A.4).

5.4. Experimental Results

As stated in the introduction, goal of this thesis is to investigate whether the bi-criteria shortest path problem can be efficiently solved in parallel. More specifically, we want to evaluate whether our `paPaSearch` implementation is efficient on modern shared memory multiprocessors.

The `paPaSearch` algorithm explores all available Pareto optimal paths in parallel. We therefore expect it to excel for large and difficult instances, i.e., instances with uncorrelated or negatively correlated objectives. These instances should not only exhibit a large number of paths to be explored simultaneously, but should also lead to larger and therefore more efficient bulk updates (in terms of temporal and spatial locality).

With the goal of finding a break-even point where parallel processing becomes worthwhile, we perform a controlled evaluation of grids with different sizes and correlations. We use sensor networks to test the dependence on the node degree, and furthermore, rely on road networks to gauge the performance on more realistic graphs. For details on the number of Pareto optima for these different instances, see Figure 5.3 and Table 5.1.

In the following, `paPaSearch (p8)` refers to parallel Pareto search algorithm using eight threads. Its sequential counterpart is referred to as `paPaSearch (seq)`. We compare both to the classic sequential competitor `LSetClassic`. All reported timing values are averages of the elapsed wall-clock time of an otherwise unloaded machine. For grids and sensor networks, all measurements are repeated on ten different instances. Measurements for road networks are repeated three times.

Performance.

In general, the results confirm the practicality of our parallel `paPaSearch` implementation. In Figures 5.7 and 5.8, we establish the performance effect of different correlations of objectives and different grid sizes. As it turns out, the parallel `paPaSearch` implementation is always faster than `LSetClassic` except for small grids with high correlations.

The parallel algorithm excels at large and difficult instances as expected. On the other hand, `paPaSearch` is clearly outperformed by `LSetClassic` for instances with highly correlated objectives. With only few labels being worked on per iteration, the overhead of `paPaSearch` compared to `LSetClassic` becomes significant. For example, while `LSet-Classic` extracts labels using a fast binary heap and can perform many dominance checks

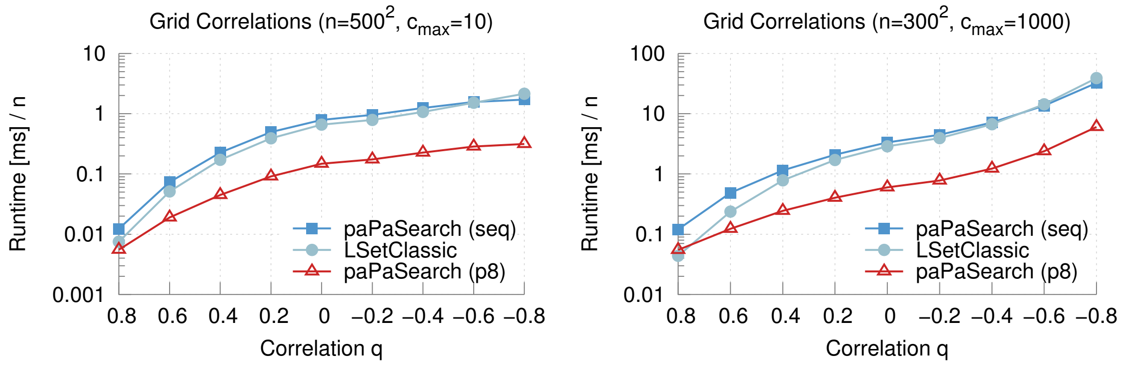


Figure 5.7.: Performance of grids with varying correlations between the two objectives. Across the board, the parallel implementation `paPaSearch (p8)` is the fastest approach. `LSetClassic` is only faster for grids with a very high correlation. Similar plots for other grid sizes are available in the appendix (Figure A.3).

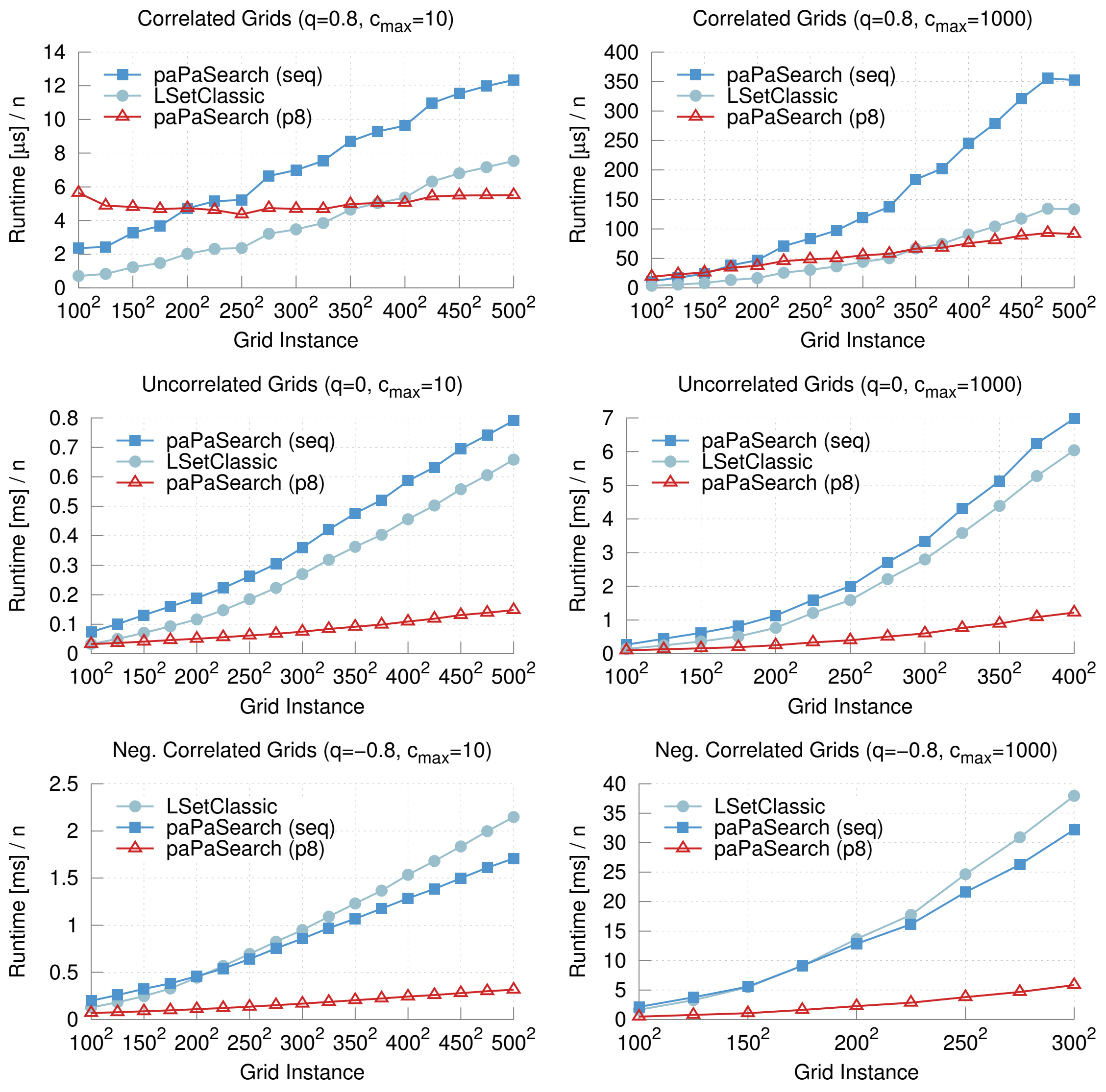


Figure 5.8.: Performance for grids of different sizes. The experiments consider the combination of three fixed correlations ($q = 0.8$, $q = 0$, and $q = -0.8$), and two different maximal edge costs ($c_{max} = 10$ and $c_{max} = 1000$). We observe that the parallel implementation `paPaSearch (p8)` is the fastest approach, except for very small grids with a high correlation of objectives. Furthermore, its sequential counterpart is competitive to `LSetClassic` for problem instances with uncorrelated and negatively correlated objectives. For the latter correlation, it is even slightly faster than the classic alternative.

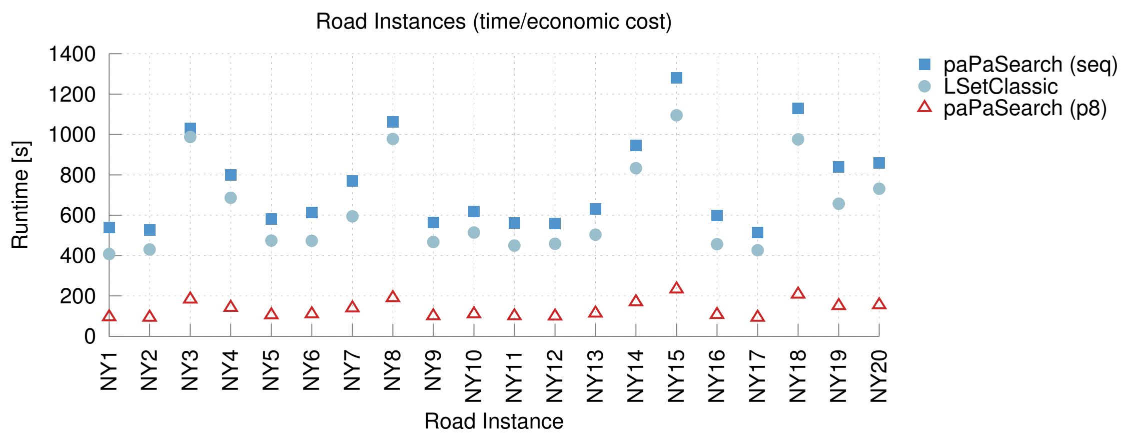


Figure 5.9.: Road instances of Machuca and Mandow [36] with uncorrelated objectives (time/economic cost) representing a real world application. The experiment confirms the results for large uncorrelated grids (see Figure 5.8).

in constant time, the `paPaSearch` operations (sorting, scanning leaves of B-trees, ...) suffer from significant higher constant factors, which can only be amortized by working on a large number of labels.

The results for the realistic road instances are very similar to respective grid instances. Difficult road instances with uncorrelated objectives are presented in Figure 5.9. Road instances with highly correlated objectives can be found in Appendix A. The experiments for the sensor networks (also available in the appendix) furthermore confirm that `paPaSearch` can also handle higher node degrees.

Speedup and Component Analysis.

Figure 5.10 presents the speedup of the parallel `paPaSearch` implementation over either `paPaSearch (seq)` or `LSetClassic`, depending on which is faster for the specific instance. We observe absolute speedups of four to five for eight threads and also a maximal speedup of at least seven.

To understand the behavior and speedup of `paPaSearch`, we analyze the different components of our implementation. Figure 5.11 shows the relative runtime of the important algorithm steps for increasing number of threads. For the sequential variant and the parallel variant using one thread, updating label sets is the most expensive step, followed by both Pareto queue operations. The relative runtime of these steps is not constant for increasing number of threads, which implies they do not scale equally. Figure 5.12 presents the corresponding speedups, which we analyze in the following.

The speedup for sorting is rather low. Probably, we do not sort enough elements for the used `tbb::parallel_sort` to scale well. The speedups for sorting updates are slightly worse than the ones for sorting candidates. We attribute this to the slightly fewer number of elements being sorted, and secondly, to the tuned sequential sort operation used for updates in the sequential case (see Section 4.3 for details on the sequential merge shortcut which does not apply in the parallel case).

The Pareto queue update procedure achieves a maximal speedup of about eight. We believe it is *memory-bound*, because even with both sockets, the system has only eight memory channels. Large amounts of data can therefore only by transferred from and to main memory with a speedup of about eight. The speedup for the extraction of Pareto minima is slightly higher than the number of available memory channels. It profits from parts of the tree still being cached after the preceding bulk update.

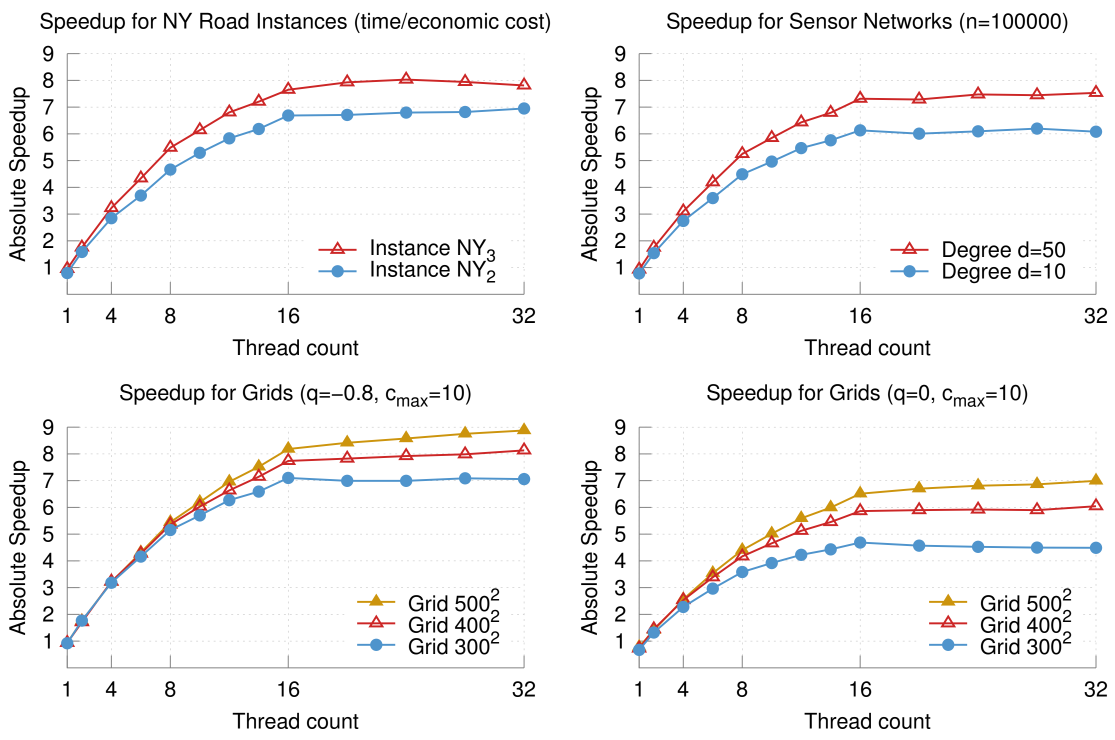


Figure 5.10.: Absolute speedup of the parallel Pareto search algorithm for different instances. The speedup is reported over either `paPaSearch (seq)` or `LSetClassic`, depending on which is faster for the specific instance. For the largest instances, the maximal achieved speedup is always about seven or eight. The speedup for smaller instances is lower, because they feature fewer Pareto optimal paths to be explored in parallel. This specific number depends on the configuration of c_{max} and q. We further observe that the speedup does not increase for thread counts larger than the physical number of cores.

In contrast to the other components, the label set update method continues to scale with increasing number of threads and even with hyper threading (i.e., more than 16 threads). Updating a label set requires a lexicographic sort of the corresponding candidate labels. This sort operation constitutes local cache-based processing, which does not require data from the main memory. Compared to the Pareto queue operations, there is therefore less pressure on the memory bandwidth, i.e., higher speedups become possible.

We conclude that the speedup of our `paPaSearch` implemented is mainly limited by the sorting procedures and the number of memory channels.

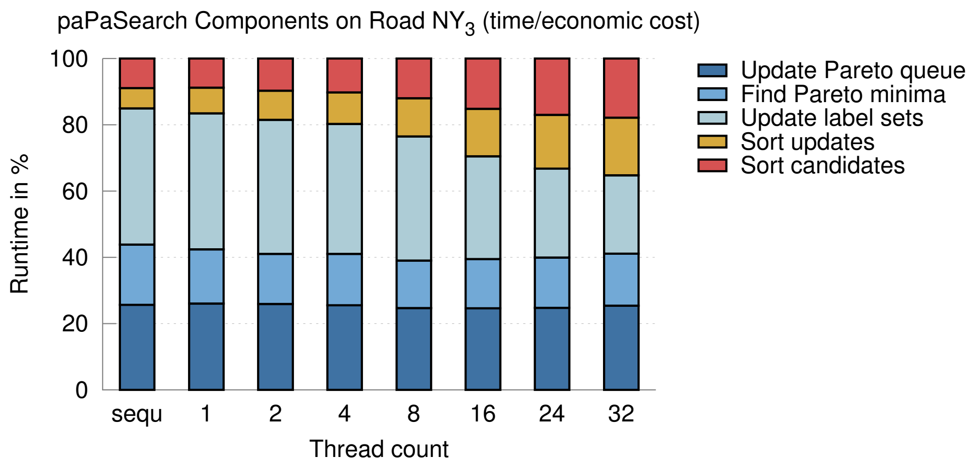


Figure 5.11.: Share of the `paPaSearch` components on the total runtime. For an explanation of shifting proportions for the increasing number of threads, see Figure 5.12 and the accompanying description in this section.

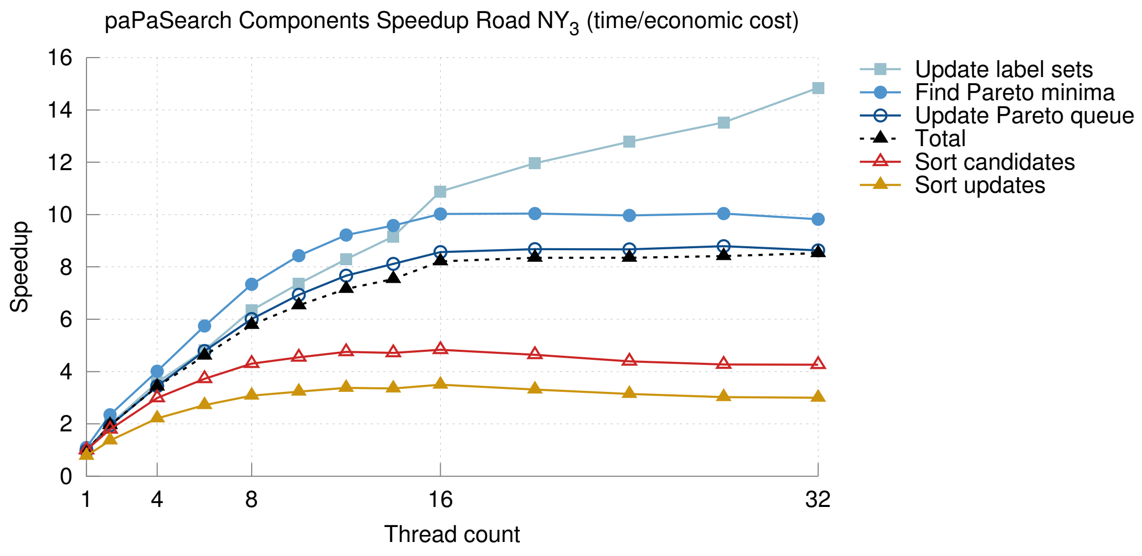


Figure 5.12.: Relative speedup of the `paPaSearch` components for increasing number of threads. The total speedup is bound by the sorting routines and the number of memory channels in the system. For a more details, see the description in this section.

6. Conclusion

This thesis presented a bi-criteria implementation of the multi-criteria shortest path algorithm of Sanders and Mandow [60], the parallel Pareto search algorithm.

We compared our implementation to a tuned classic label setting algorithm. The results indicate that the parallel Pareto search algorithm is competitive for large and difficult instances when run sequentially. Furthermore, it is shown to exhibit significant speedups when run in parallel. According to our analysis, this speedup is bound by the performance of parallel shared memory sorting procedures and the systems memory bandwidth. We have therefore demonstrated that parallel bi-criteria shortest path search is practical on modern shared memory multiprocessors.

As an additional contribution of independent interest, we presented parallel bulk updates for weight-balanced B-trees. We used this technique as the basis for a cache-efficient implementation of important data structures of the parallel bi-criteria algorithm. The performance of the latter is hereby improved greatly; Sanders and Mandow [60] proposed to implement these data structures based on parallel binary trees, which are significantly slower than our solution. As future work, it seems worthwhile to concentrate on representing our B-tree more compactly in order to use the precious memory bandwidth more efficiently. For example, we might apply well-established B-tree compression techniques (for a survey, see [22]).

Regardless of the good results for difficult instances, our implementation does not perform well for instances with highly correlated objectives. Featuring only few Pareto optimal paths, these instances are normally considered easy. However, this also implies that only few paths can be explored in parallel. We might be able to improve this situation to our advantage by applying the *delta-stepping* [42] idea to the Pareto queue, i.e., increasing the number of extracted labels per iteration. This should lead to more parallelizable work and probably fewer iterations in total. By increasing the size of bulk updates and therefore the temporal and spatial locality of our algorithm, delta stepping might also help to increase the performance for more difficult instances.

Finally, the algorithm extensions proposed by Sanders and Mandow [60] are other promising candidates for future work. These extensions include single target search (one-to-one bi-criteria shortest path search) and the adaptation of the algorithm to work with more than two objectives.

7. Appendix

A. Parallel Pareto Search

Figures A.1, A.2, and A.3 present additional experimental results of the `paPaSearch` algorithm.

Furthermore, we present label set modifications (Figure A.4) and Pareto queue modifications (Figure A.5) of the parallel Pareto search algorithm [60] for different road and grid instances. All presented instances are exemplary, i.e., we consider just a single random grid per configuration and present only two representative road instances. Details on the different instance types can be found in section 5.2.

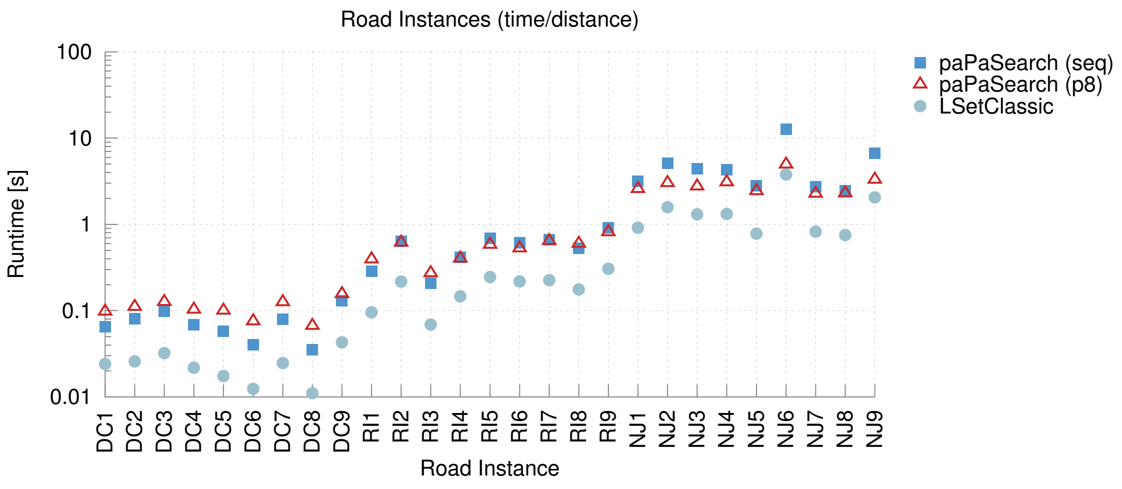


Figure A.1.: Road instances of Raith and Ehrgott [54] with highly correlated objectives (time/distance). This experiment confirms the results for grid instances (see Figures 5.8 and 5.7), i.e., `paPaSearch` is not competitive for highly correlated objectives.

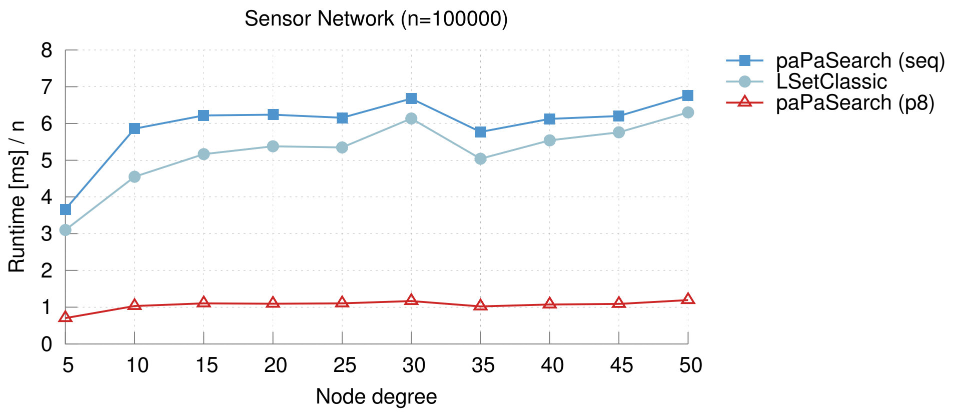


Figure A.2.: Sensor networks showing that `paPaSearch` can also handle higher out degrees with results similar to the road and grid instances.

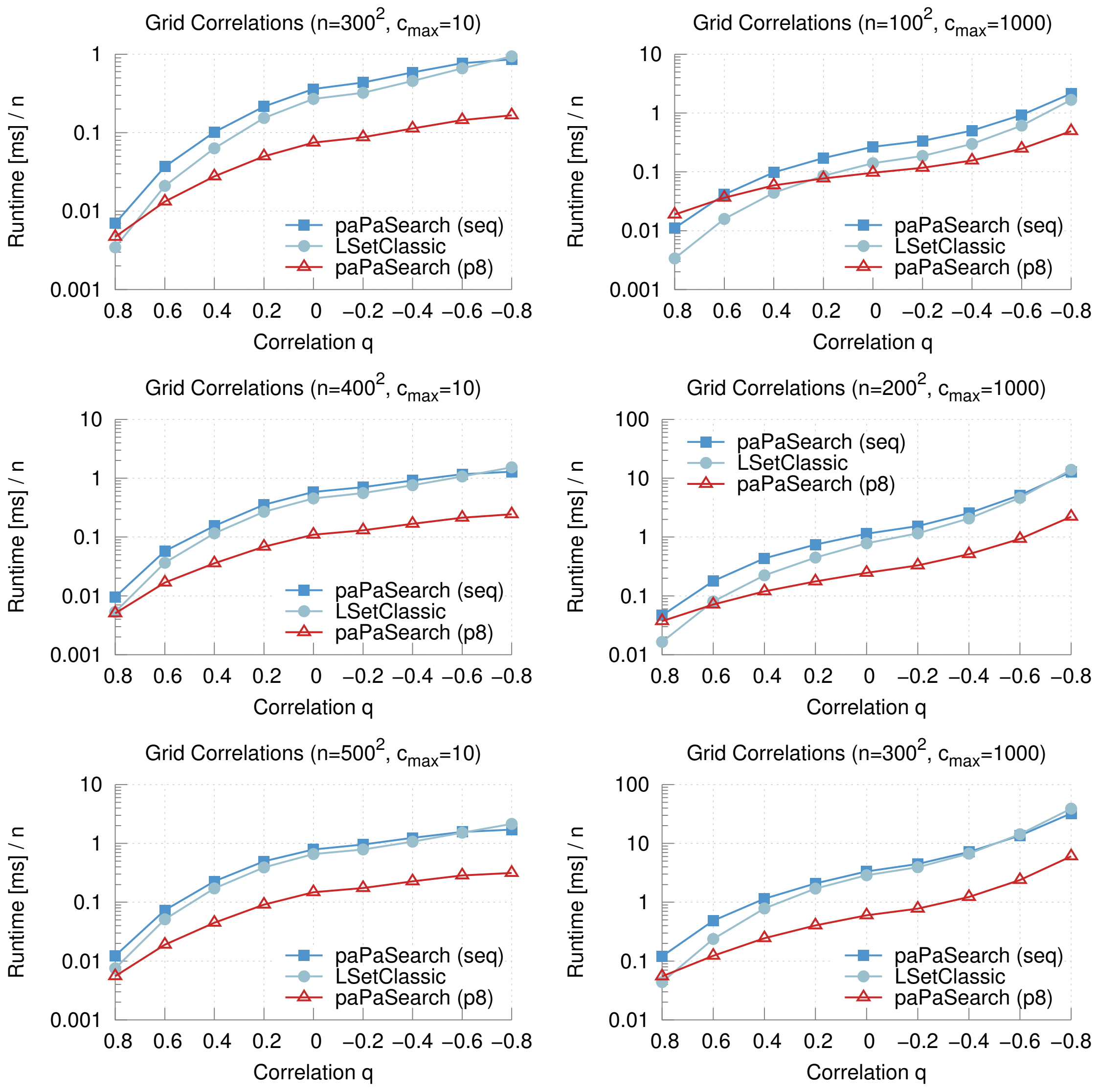


Figure A.3.: Performance of grids with varying correlations between the two objectives. The experiments consider grids of two different sizes and with two different maximal edge costs ($c_{max} = 10$ and $c_{max} = 1000$). Across the board, the parallel implementation `paPaSearch (p8)` is the fastest approach. `LSetClassic` is only faster for small grids with high correlation.

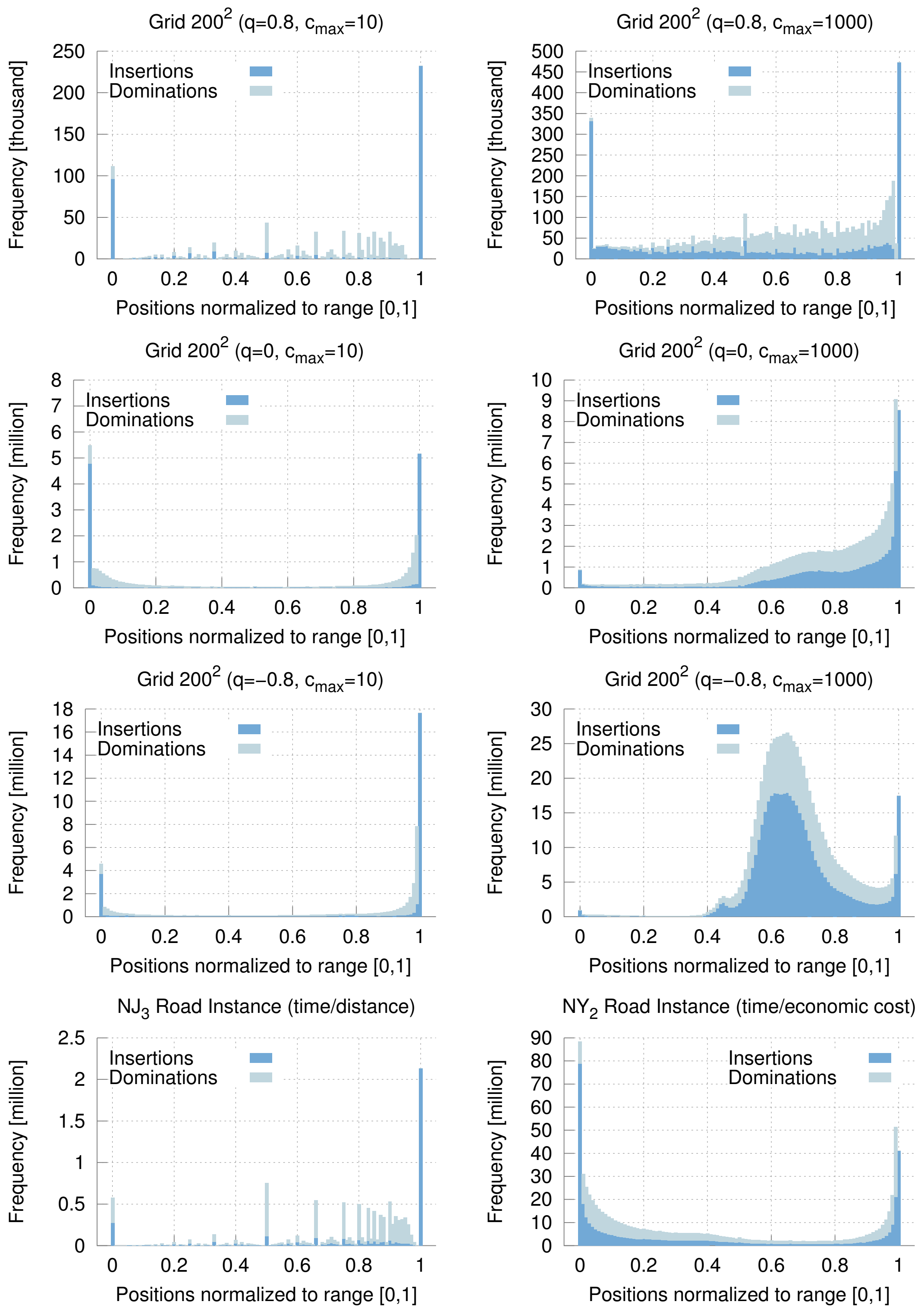


Figure A.4.: Position within a label set where candidate labels are either dominated or inserted by our parallel labeling algorithm.

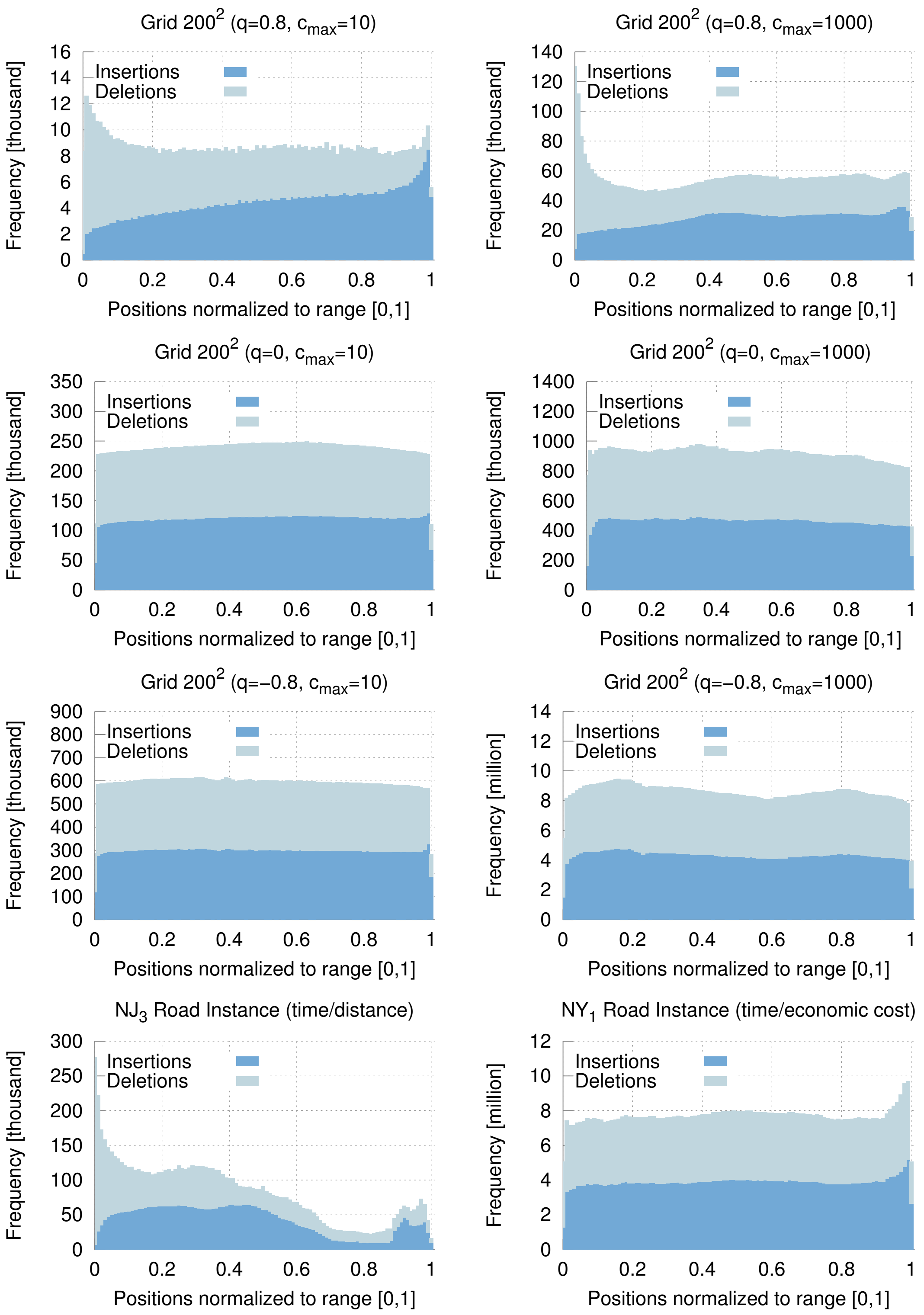


Figure A.5.: Position within the Pareto queue where tentative labels are either inserted or removed. Deletions correspond to the positions where Pareto minima are found.

B. Classic Bi-Criteria Label Setting

Figure B.6 details the label set access pattern for road instances. The Algorithms 7.1 and 7.2 show the corresponding label set implementation. We use a binary search for x-`predecessor` and scanning for our y-`predecessor` implementation. To prevent bound checks within these functions, we add sentinels of lexicographically minimal and maximal labels to each label set (not shown here). Table B.1 furthermore shows the grid instances used to evaluate this implementation.

Name	$h \times w$	Nodes	Edges	Avg. Pareto optima
G1	30 × 40	1 202	4 720	35.24
G2	20 × 80	1 602	6 240	88.92
G3	50 × 90	4 502	17 820	112.33
G4	90 × 50	4 502	17 900	50.33
G5	50 × 200	10 002	39 600	286.08
G6	200 × 50	10 002	39 900	50.17
G7	100 × 150	15 002	59 700	188.33
G8	150 × 100	15 002	59 800	123.33
G9	100 × 200	20 002	79 600	273.75
G10	200 × 100	20 002	79 800	124.00
G11	200 × 150	30 002	79 800	205.17
G12	50 × 50	10 002	39 600	50.83
G13	100 × 100	10 002	39 800	115.67
G14	200 × 200	40 002	159 600	279.67
G15	2 450 × 2	4 902	19 596	4.75
G16	1 225 × 4	4 902	19 592	7.83
G17	612 × 8	4 898	19 586	8.75
G18	288 × 17	4 898	19 550	17.33
G19	196 × 25	4 902	19 550	22.33
G20	140 × 35	4 902	19 530	31.75
G21	111 × 44	4 886	19 448	48.25
G22	92 × 53	4 878	19 398	58.08
G23	79 × 62	4 900	19 468	69.33
G24	70 × 70	4 902	19 460	78.33
G25	62 × 79	4 900	19 343	95.00
G26	53 × 92	4 878	19 320	109.92
G27	44 × 111	4 886	19 314	135.67
G28	35 × 140	4 902	19 320	185.08
G29	25 × 196	4 902	19 208	260.33
G30	17 × 288	4 898	19 008	412.50
G31	8 × 612	4 898	18 360	815.92
G32	4 × 1 225	4 902	17 150	1 376.25
G33	2 × 2 450	4 902	19 596	1 574.83

Table B.1.: Characteristics of the grid instances of Raith and Ehrgott [54] used in the experiment presented in Figure 5.2. The number of optima refers to the number of optimal paths from the start at one side of the grid to the end node at the other side, not counting Pareto optimal paths to other nodes.

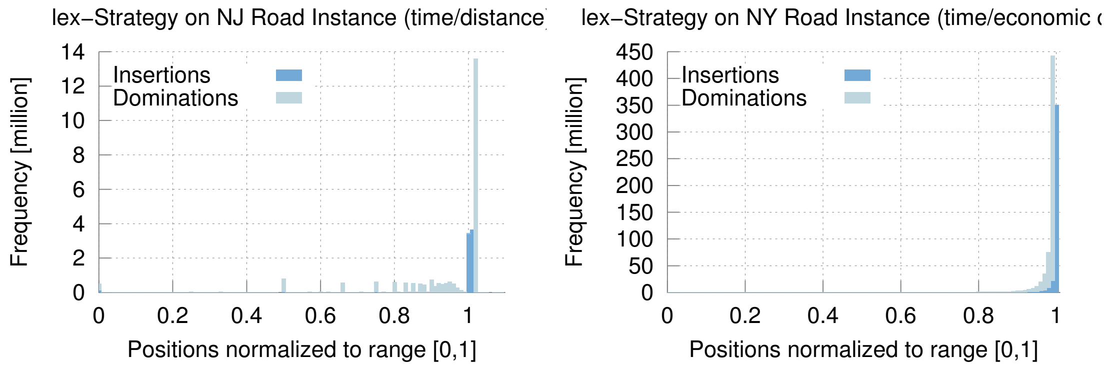


Figure B.6.: Classic bi-criteria label setting: Label set operations for the `lex` label selection strategy on road instances.

Algorithm 7.1: Dominance Check

```
Input: Label set $L$ of size $n$, Candidate label $l$
Output: true if $l$ is dominated by any label in $L$, false otherwise
1 if $L[n]_x < l_x$ then
2     return $L[n]_y \leq l_y$
3 else
4     $k \leftarrow x$-predecessor$(L, l)$      // Position of largest labels with smaller x-coord.
5     if $L[k]_y \leq l_y$ then
6         return true                         // l is dominated by its predecessor
7     else
8         $k++$                   // Move to element with greater or equal x-coordinate
9         return $(L[k]_x = l_x$ and $L[k]_y \leq l_y)$
```

Algorithm 7.2: Label Set Update

```
Input: Label set $L$ of size $n$, Candidate label $l$
1 if $l$ is not dominated by $L[v]$ then
2     Let $k$ be the insertion position computed by the dominance check
3     $j \leftarrow y$-predecessor$(L, l)$            // Position of the first non-dominated label
4     if $k = j$ then
5         Insert into $L$ at position $k$. Shift succeeding labels
6     else
7         Replace label at position $k$ with $l$
8         Delete other labels in the range $[k+1, j)$
9     Update the heap $Q$ according to changes in $L$
```

Bibliography

[1] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. ISSN 0178-4617. doi: 10.1007/s00453-003-1021-x. URL `http://dx.doi.org/10.1007/s00453-003-1021-x`.

[2] L. Arge. External-memory geometric data structures. *Lecture Notes on I/O-algorithms*, 2006. URL `http://daimi.au.dk/~large/ioS06/ionotes.pdf`.

[3] L. Arge and J. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003. URL `http://dx.doi.org/10.1137/S009753970240481X`.

[4] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. pages 197–206, 2008. doi: 10.1145/1378533.1378573. URL `http://doi.acm.org/10.1145/1378533.1378573`.

[5] D. Bader, B. Moret, and P. Sanders. Algorithm engineering for parallel computation. In R. Fleischer, B. Moret, and E. Schmidt, editors, *Experimental Algorithmics*, volume 2547 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-00346-5. doi: 10.1007/3-540-36383-1_1. URL `http://dx.doi.org/10.1007/3-540-36383-1_1`.

[6] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 399–409. IEEE Comput. Soc, 2000. ISBN 0-7695-0850-2. doi: 10.1109/SFCS.2000.892128. URL `http://dx.doi.org/10.1109/SFCS.2000.892128`.

[7] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 228–237, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1074009. URL `http://doi.acm.org/10.1145/1073970.1074009`.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL `http://doi.acm.org/10.1145/324133.324234`.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. URL `http://doi.acm.org/10.1145/209936.209958`.

[10] G. Brodal and K. Tsakalidis. Dynamic planar range maxima queries. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22005-0. doi: 10.1007/978-3-642-22006-7_22. URL `http://dx.doi.org/10.1007/978-3-642-22006-7_22`.

[11] J. Brumbaugh-Smith and D. Shier. An empirical investigation of some bicriterion shortest path algorithms. *European Journal of Operational Research*, 43(2):216 – 224, 1989. ISSN 0377-2217. doi: 10.1016/0377-2217(89)90215-4. URL `http://www.sciencedirect.com/science/article/pii/0377221789902154`.

[12] M. Carey and C. Thompson. An efficient implementation of search trees on [lg n + 1] processors. *Computers, IEEE Transactions on*, C-33(11):1038–1041, 1984. ISSN 0018-9340. doi: 10.1109/TC.1984.1676379.

[13] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. pages 235–246, 2001. doi: 10.1145/375663.375688. URL `http://doi.acm.org/10.1145/375663.375688`.

[14] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301633. URL `http://doi.acm.org/10.1145/301618.301633`.

[15] J. Clímaco, J. Craveirinha, and M. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Networks*, 41(4):206–220, 2003. URL `http://onlinelibrary.wiley.com/doi/10.1002/net.10073/abstract`.

[16] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. ISSN 0360-0300. doi: 10.1145/356770.356776. URL `http://doi.acm.org/10.1145/356770.356776`.

[17] S. Demeyer, J. Goedgebeur, P. Audenaert, M. Pickavet, and P. Demeester. Speeding up martins' algorithm for multiple objective shortest path problems. *4OR*, pages 1–26, 2013. ISSN 1619-4500. doi: 10.1007/s10288-013-0232-5. URL `http://dx.doi.org/10.1007/s10288-013-0232-5`.

[18] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL `http://dx.doi.org/10.1007/BF01386390`.

[19] L. Frias and J. Singler. Parallelization of bulk operations for stl dictionaries. In L. Bougé, M. Forsell, J. Träff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, editors, *Euro-Par 2007 Workshops: Parallel Processing*, volume 4854 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78472-2. doi: 10.1007/978-3-540-78474-6_8. URL `http://dx.doi.org/10.1007/978-3-540-78474-6_8`.

[20] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979.

[21] R. G. Garroppo, S. Giordano, and L. Tavanti. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Computer Networks*, 54(17): 3081 – 3107, 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2010.05.017. URL `http://www.sciencedirect.com/science/article/pii/S1389128610001659`.

[22] G. Graefe and P. Larson. B-tree indexes and CPU caches. In *Proceedings 17th International Conference on Data Engineering*, pages 349–358. IEEE Comput. Soc, 2001. ISBN 0-7695-1001-9. doi: 10.1109/ICDE.2001.914847.

[23] F. Guerriero and R. Musmanno. Label correcting methods to solve multicriteria shortest path problems. *Journal of Optimization Theory and Applications*, 111(3): 589–613, 2001. ISSN 0022-3239. doi: 10.1023/A:1012602011914. URL `http://dx.doi.org/10.1023/A%3A1012602011914`.

[24] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21, 1978. doi: 10.1109/SFCS.1978.3.

[25] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

[26] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 283–294, New York, NY, USA, 2003. ACM. ISBN 1-58113-664-1. doi: 10.1145/781027.781063. URL `http://doi.acm.org/10.1145/781027.781063`.

[27] P. Hansen. Bicriterion path problems. In *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer Berlin Heidelberg, 1980. ISBN 978-3-540-09963-5. doi: 10.1007/978-3-642-48782-8_9. URL `http://dx.doi.org/10.1007/978-3-642-48782-8_9`.

[28] L. Higham and E. Schenk. Maintaining B-trees on an EREW PRAM. *Journal of parallel and distributed computing*, 1994. URL `http://www.sciencedirect.com/science/article/pii/S0743731584710926`.

[29] M. Iori, S. Martello, and D. Pretolani. An aggregate label setting policy for the multi-objective shortest path problem. *European Journal of Operational Research*, 207(3):1489 – 1496, 2010. ISSN 0377-2217. doi: 10.1016/j.ejor.2010.06.035. URL `http://www.sciencedirect.com/science/article/pii/S0377221710004741`.

[30] N. Jozefowiez, F. Semet, and E.-G. Talbi. Multi-objective vehicle routing problems. *European Journal of Operational Research*, 189(2):293 – 309, 2008. ISSN 0377-2217. doi: 10.1016/j.ejor.2007.05.055. URL `http://www.sciencedirect.com/science/article/pii/S0377221707005498`.

[31] A. Kukanov and M. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(04), 2007. doi: 10.1535/itj.1104.05.

[32] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, 1994.

[33] K. S. Larsen. Relaxed multi-way trees with group updates. *Journal of Computer and System Sciences*, 66(4):657–670, June 2003. ISSN 00220000. doi: 10.1016/S0022-0000(03)00027-8. URL `http://linkinghub.elsevier.com/retrieve/pii/S0022000003000278`.

[34] T. Lilja, R. Saikkonen, S. Sippu, and E. Soisalon-Soininen. Online bulk deletion. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 956–965, 2007. doi: 10.1109/ICDE.2007.368954.

[35] E. Machuca. *An Analysis of Some Algorithms and Heuristics for Multiobjective Graph Search.* PhD thesis, University of Malaga, 2012. URL `http://riuma.uma.es/xmlui/handle/10630/5072`.

[36] E. Machuca and L. Mandow. Multiobjective heuristic search in road maps. *Expert Systems with Applications*, 39(7):6435 – 6445, 2012. ISSN 0957-4174. doi: 10.1016/j.eswa.2011.12.022. URL `http://www.sciencedirect.com/science/article/pii/S0957417411016939`.

[37] E. Machuca, L. Mandow, J. Pérez de la Cruz, and A. Ruiz-Sepulveda. A comparison of heuristic best-first algorithms for bicriterion shortest path problems. *European Journal of Operational Research*, 217(1):44–53, Feb. 2012. ISSN 03772217. doi: 10.1016/j.ejor.2011.08.030. URL `http://linkinghub.elsevier.com/retrieve/pii/S0377221711008010`.

[38] L. Malmi and E. Soisalon-Soininen. Group updates for relaxed height-balanced trees. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '99*, pages 358–367, New York, New York, USA, 1999. ACM Press. ISBN 1581130627. doi: 10.1145/303976.304011.

[39] L. Mandow and J. Cruz. A memory-efficient search strategy for multiobjective shortest path problems. In *KI 2009: Advances in Artificial Intelligence*, volume 5803 of *Lecture Notes in Computer Science*, pages 25–32. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04616-2. doi: 10.1007/978-3-642-04617-9_4. URL `http://dx.doi.org/10.1007/978-3-642-04617-9_4`.

[40] E. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984. URL `http://www.sciencedirect.com/science/article/pii/0377221784900778`.

[41] K. Mehlhorn and P. Sanders. *Algorithms and data structures: The basic toolbox.* Springer, 2008.

[42] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152, 2003. ISSN 0196-6774. doi: 10.1016/S0196-6774(03)00076-2. URL `http://www.sciencedirect.com/science/article/pii/S0196677403000762`.

[43] J. Mote, I. Murthy, and D. Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53:81–92, 1991. URL `http://www.sciencedirect.com/science/article/pii/037722179190094C`.

[44] M. Müller-Hannemann and K. Weihe. Pareto shortest paths is often feasible in practice. In G. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42500-7. doi: 10.1007/3-540-44688-5_15. URL `http://dx.doi.org/10.1007/3-540-44688-5_15`.

[45] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '87, pages 170–176, New York, NY, USA, 1987. ACM. ISBN 0-89791-223-3. doi: 10.1145/28659.28677. URL `http://doi.acm.org/10.1145/28659.28677`.

[46] M. Overmars. *The design of dynamic data structures.* Springer Verlag, 1983. ISBN 354012330X.

[47] J. Paixão and J. Santos. Labelling methods for the general case of the multi-objective shortest path problem-a computational study. pages 1–23, 2007.

[48] H. Park and K. Park. Parallel algorithms for red-black trees. *Theoretical Computer Science*, 262(1-2):415–435, July 2001. ISSN 03043975. doi: 10.1016/S0304-3975(00)00287-5. URL `http://linkinghub.elsevier.com/retrieve/pii/S0304397500002875`.

[49] H. Park, K. Park, and Y. Cho. Deleting keys of b-trees in parallel. *Journal of Parallel and Distributed Computing*, 64(9):1041 – 1050, 2004. ISSN 0743-7315. doi: 10.1016/

j.jpdc.2004.06.005. URL http://www.sciencedirect.com/science/article/pii/S0743731504001091.

[50] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In J. Diaz, editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 597–609. Springer Berlin Heidelberg, 1983. ISBN 978-3-540-12317-0. doi: 10.1007/BFb0036940. URL http://dx.doi.org/10.1007/BFb0036940.

[51] K. Pollari-Malmi and E. Soisalon-Soininen. Concurrency control and i/o-optimality in bulk insertion. In A. Apostolico and M. Melucci, editors, *String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 161–170. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23210-0. doi: 10.1007/978-3-540-30213-1_24. URL http://dx.doi.org/10.1007/978-3-540-30213-1_24.

[52] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996. ISSN 10414347. doi: 10.1109/69.553166. URL http://dx.doi.org/10.1109/69.553166.

[53] A. Raith. Speed-up of Labelling Algorithms for Biobjective Shortest Path Problems. *Proceedings of the 45th annual conference of the ORSNZ. Auckland, New Zealand*, pages 313–322, 2010.

[54] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, Apr. 2009. ISSN 03050548. doi: 10.1016/j.cor.2008.02.002. URL http://linkinghub.elsevier.com/retrieve/pii/S0305054808000233.

[55] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. pages 0–17, 1998. URL http://hdl.handle.net/10022/AC:P:29336.

[56] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335449. URL http://doi.acm.org/10.1145/342009.335449.

[57] R. Saikkonen. Bulk updates and cache sensitivity in search trees. 2009. URL https://aaltodoc.aalto.fi/handle/123456789/4649.

[58] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics (JEA)*, 5, Dec. 2000. ISSN 1084-6654. doi: 10.1145/351827.384249. URL http://doi.acm.org/10.1145/351827.384249.

[59] P. Sanders. Algorithm engineering – an attempt at a definition. In S. Albers, H. Alt, and S. Näher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03455-8. doi: 10.1007/978-3-642-03456-5_22. URL http://dx.doi.org/10.1007/978-3-642-03456-5_22.

[60] P. Sanders and L. Mandow. Parallel label-setting multi-objective shortest path search. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS2013)*. IEEE Computer Society, May 2013.

[61] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[62] N. Sitchinava and N. Zeh. A parallel buffer tree. *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12*, page 214, 2012. doi: 10.1145/2312005.2312046. URL `http://dl.acm.org/citation.cfm?doid=2312005.2312046`.

[63] A. Skriver and K. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507–524, May 2000. ISSN 03050548. doi: 10.1016/S0305-0548(99)00037-4.

[64] A. J. Skriver. A classification of bicriterion shortest path (bsp) algorithms. *Asia Pacific Journal of Operational Research*, 17(2):199–212, 2000.

[65] P. Vincke. Problemes multicriteres. *Universite Libre de Bruxelles. Centre d'Etudes de Recherche Operationnelle. Cahiers*, 16(4), 1974.

[66] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 160–169. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23396-8. doi: 10.1007/978-3-642-23397-5_16. URL `http://dx.doi.org/10.1007/978-3-642-23397-5_16`.

[67] R. Wein. Efficient implementation of red-black trees with split and catenate operations. pages 1–11, 2005.