Parallel Label-Setting Multi-Objective Shortest Path Search

Peter Sanders
Institute for Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: sanders@kit.edu

Lawrence Mandow
University of Málaga
Málaga, Spain
Email: lawrence@lcc.uma.es

***Abstract*—We present a parallel algorithm for finding all Pareto optimal paths from a specified source in a graph. The algorithm is label-setting, i.e., it only performs work on distance labels that are optimal. The main result is that the added complexity when going from one to multiple objectives is completely parallelizable. The algorithm is based on a multi-objective generalization of a priority queue. Such a *Pareto queue* can be efficiently implemented for two dimensions. Surprisingly, the parallel biobjective approach yields an algorithm performing asymptotically *less* work than the previous sequential algorithms. We also discuss generalizations for $d \geq 3$ objective functions and for single target search.**

I. Introduction

Finding shortest paths in graphs is a classical optimization problem with a large number of applications. Unfortunately, it is sometimes difficult to fix the objective function. Often there are several relevant objectives and it is not clear how they should be combined. For example, when you plan a car trip you are likely to be interested in both fast and fuel-efficient routes but the best compromise depends on how much you are in a hurry. *Multi-objective* search is an attractive model here – we are searching for *Pareto optimal* solutions, i.e., paths that are not dominated by any other path (worse with respect to all objective functions). Multi-objective search is NP hard even for $d = 2$ objective functions [9], i.e., even if we only want to solve the decision problem whether there exists a path between two nodes whose length is below some threshold with respect to both objectives, it is unlikely that we will ever find a polynomial time algorithm for this problem. For the optimization problem, the output can also have exponential size. A classical example is a directed path $v_1 \Rightarrow v_2 \Rightarrow \cdots \Rightarrow v_n$ with two parallel edges (v_i, v_{i+1}) between successive nodes where one of these edges has weight $(0, 2^i)$ and the other has weight $(2^i, 0)$.[1] Still, multi-objective search can be practical at least for small d and not too large graphs, e.g., [20]. In particular, there are *label-setting* algorithms for the problem that are near optimal in the sense that work is only performed on data that in some sense is part of the output (refer to Section II for a more detailed discussion).

[1]One gets a similar example without parallel edges by replacing the edges by two-edge paths.

We are not aware of attempts to parallelize multi-objective path search. This is surprising since multi-objective search is expensive and since parallel processors are now ubiquitously available, i.e., any computationally demanding problem has to be parallelized in order to run efficiently on modern hardware. This paper is a first step at closing this gap. The result is surprisingly positive at least for $d = 2$: There is a parallel label setting algorithm and *all* the additional work to the single-objective problem is parallelizable. Moreover, the overhead for parallelizaton is *negative* in an asymptotic sense – when the overall work is superpolynomial in the graph size then the parallel algorithm performs less work than the corresponding sequential algorithm. Another positive aspect of our result is that it shows that multi-objective search is parallelizable in contrast to the single-objective search which is notoriously hard to parallelize, in particular when we want a label-setting algorithm and when the paths are long.

Section II introduces the problem and the algorithmic foundations in more detail followed by additional related work in Section III. Section IV derives the basic algorithm which works for any number d of objectives and finds all Pareto optimal paths from a given source node to all other nodes. Section V concretizes this algorithm for the case $d = 2$ parallelizing some basic results from computational geometry. Section VI extends the results to goal-directed single-target search followed by a short discussion of more than two objectives in Section VII. Section VIII summarizes the results and discusses possible future work.

II. Preliminaries

We consider a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ with d-dimensional weight function $\vec{c} : E \to \mathbb{R}^d_{\geq 0}$.[2] The weight function is extended to paths by summing weights component by component. Each of the coordinates is also called one of the d *objectives* of our search. Value x_i denotes the i-th component of a vector $\vec{x}$. $\vec{x} \neq \vec{y}$ dominates $\vec{y}$ ($\vec{x} \prec \vec{y}$) if and only if $\forall i \in 1..d : x_i \leq y_i$. Given a set of points M, elements of M that are not dominated by any element in

[2]For graphs that do not have negative weight cycles with respect to any coordinate, one can lift the requirement of nonnegative edge weights by introducing vertex potentials in the standard way [18] – using a single objective search for each dimension which is independent of the source node.

M are called *Pareto optimal*. A set consisting solely of Pareto optimal points is a *Pareto set*.

Given a source node s, we are looking for Pareto optimal paths from s to all other nodes v (*one-to-all* problem), i.e., paths whose length is not dominated by any other path from s to v. In Section VI we discuss the case when we are interested only in a single target node t (*one-to-one* problem).

(Multi-Objective) shortest path search algorithms are in many ways generalizations of single-objective algorithms. They compute labels of nodes. A label $(v, \vec{\ell}) \in V \times \mathbb{R}^d_{\geq 0}$ represents an s–v-path with length vector $\vec{\ell}$. Labels are updated by *edge relaxations*: Given a node u with label $\vec{\ell}$ and an edge (u, v) with weight $\vec{x}$, an edge relaxation computes a candidate label $\vec{\ell'} = \vec{\ell} + \vec{x}$ for node v. Furthermore, the relaxation checks whether $\vec{\ell'}$ is new and not dominated by any previously computed labels for v. In this case, $\vec{\ell'}$ is added to the label set $L[v]$ of v and labels of v dominated by $\vec{\ell'}$ are removed (pruned). In order to be able to reconstruct paths, the new label can be augmented with a parent reference to the label $(u, \vec{\ell})$. A search algorithm is called *label-setting* if it only performs relaxations when $(u, \vec{\ell})$ actually represents a Pareto optimal path. Algorithms which may also relax on suboptimal labels are called *label-correcting*. Usually edge relaxations are grouped into label *scanning* operations where all edges out of u are relaxed. For the single-objective case, Dijkstra's algorithm is a label setting algorithm. It keeps unscanned labels in a priority queue and performs n iterations always scanning the node with the smallest label.

The classical sequential label setting-algorithms for multi-objective search [17], [24] define a total order on labels with the property that a minimal label is Pareto optimal among all labels. For example, any positive linear combination (i.e., all coefficients are positive) or any lexicographic ordering has this property. Using this ordering we then obtain an algorithm largely analogous to Dijkstra's algorithm: Unscanned labels are kept in a priority queue. In each iteration, the minimal element of the queue is removed and scanned. Newly discovered labels from the resulting edge relaxations are inserted into the queue. Old labels which are now dominated are removed from the queue. An obvious approach to parallelization would be to remove not only one but multiple elements from the queue. However, selecting these nodes purely due to their ordering would destroy the label-setting property – the single-dimensional ordering destroys valuable information. We will therefore pursue a more careful approach working with the full-dimensional labels.

III. More Related Work

Hansen [13] describes the first label-setting algorithm with two objectives. Martins [17] extends the idea to the general multi-objective case using lexicographic ordering of labels. Tung and Chew give a similar procedure ordering labels with a linear combination of the objectives [24]. Paixão and Santos review other possible orderings [21].

Guerriero and Musmanno [10] compare the performance of label-setting and label-correcting algorithms over a set of one-to-all problems with random edge weights. Label-correcting algorithms were faster in low density graphs[3], while Martin's label-setting algorithm was faster over similar, but more dense, graphs. These authors further conclude that "parallel computing [...] represents the main goal for future developments" in this field. One important point to remember is that label-setting methods perform less scanning operations and when they are outperformed by label-correcting methods the margin is only due to a relatively small overhead in the data structures. Studies for single-objective search also indicate that more sophisticated priority queue data structures might further improve the performance of label-setting algorithms [5].

Regarding one-to-one (single target) problems, several different approaches have been proposed [23]. Raith and Ehrgott [22] report an extensive evaluation of label-setting, label-correcting, an enumerative approach, and different combinations of a *two-phase* approach. They use sets of biobjective problems over random graphs, random grids, and road networks. All successful approaches there use label-scanning as their main computational engine that can be parallelized with the algorithms presented in our paper. The first phase of the two-phase method produces a problem decomposition that can be used as an additional source of parallelism. However, it is not clear how to parallelize this decomposition in a worst-case efficient way. Regarding goal-directed search, the recent work on the NAMOA* algorithm has established the best way to introduce consistent node potentials to improve search performance in label-setting algorithms [16]. A recent computational study revealed that this approach is generally the most effective one when compared to undirected search or previous goal-directed approaches [15]. Therefore, we parallelize this algorithm in Section VI.

While we are not aware of attempts to parallelize multi-objective path search, there is some work on parallel single source algorithms. For random edge weights and low diameter networks, there is some parallelism available [6] for label-setting algorithms, but Δ-stepping, a label-correcting algorithm, works better [19]. Still, both in the worst case and for many real world networks, the shortest path problem remains difficult to parallelize beyond parallelizing the priority queue operations [3]. However, this gives only very little parallelism when only a single element can be selected. Hence, there has not yet been a successful practical parallelization.

IV. The Basic Algorithm

Previous sequential label-setting algorithms are based on the observation that *any* globally Pareto optimal label can be scanned. The starting point for this paper is that the label-setting property remains intact if *any subset* of globally Pareto optimal labels are scanned in parallel. See Figure 1 for a high level pseudo code. The algorithm maintains a set Q of node-label pairs which are locally Pareto optimal at their respective nodes. Q is initialized to $\{(s, 0^d)\}$. In each iteration, the algorithm identifies the set L of (globally)

[3]Paixão and Santos [21] report similar results on similar graphs.

Pareto optimal elements in Q, removes them from Q, and scans them. All these operations can be parallelized. Previous sequential algorithms are the special case with $|L| = 1$. Moreover, if $d = 1$, we obtain Dijkstra's algorithm. Before going into details of the parallel implementation we show basic properties.

Theorem 1: The algorithm from Figure 1 is a label-setting algorithm finding all Pareto optimal paths from s to all nodes in the network. Moreover, the number of edge relaxations performed does not depend on the strategy used for selecting subsets of Pareto optimal labels to be scanned.

The proof is largely analogous to the proof of Dijkstra's algorithm in [18]. The main generalization is in the fact that sometimes node labels play the role of nodes in the proof.

Proof: We proceed in two steps. In the first step, we show that all optimal labels are scanned exactly once. In the second step, we show that only optimal labels are ever scanned. In both steps, we argue by contradiction.

For the first step, assume the existence of a label $(v, \vec{\ell})$ that is witnessed by an optimal s–v-path $P = \langle s = v_1, v_2, \ldots, v_k = v \rangle$ with labels $\vec{\ell}_1, \ldots, \vec{\ell}_k$, but never scanned. Let $(v_i, \vec{\ell}_i)$ denote the first label on P that is never scanned. Then $i > 1$, since $(s, 0^d)$ is the first label scanned (in the first iteration). By the definition of i, $(v_{i-1}, \vec{\ell}_{i-1})$ has been scanned. When $(v_{i-1}, \vec{\ell}_{i-1})$ is scanned, $\vec{\ell}_{i-1} + \vec{c}((v_{i-1}, v_i)) = \vec{\ell}_i$ is a candidate label for v_i. Since it is optimal for v_i it is also inserted into Q and never removed as dominated. So $(v_i, \vec{\ell}_i)$ must be scanned at some point during the execution – a contradiction. For the "exactly once" part note that once a label is scanned it can never enter Q again since only new labels can enter Q.

For the second step, consider the first iteration t, when a suboptimal label $(v, \vec{\ell})$ is scanned and consider an optimal path $P = \langle s = v_1, v_2, \ldots, v_k = v \rangle$ with labels $\vec{\ell}_1, \ldots, \vec{\ell}_k \prec \vec{\ell}$. As above, let i be minimal such that v_i is not scanned before iteration t. Then $i > 1$, since $(s, 0^d)$ is the first label scanned. By the definition of i, $(v_{i-1}, \vec{\ell}_{i-1})$ was scanned before time t and hence the locally optimal candidate label $(v_i, \vec{\ell}_{i-1} + \vec{c}((v_{i-1}, v_i)) = (v_i, \vec{\ell}_i)$ was inserted into Q (or was already present there). By definition of i, this label is still present in Q at iteration t. But $\vec{\ell}_i \preceq \vec{\ell}_k \prec \vec{\ell}$ which makes it impossible that $(v, \vec{\ell})$ is removed from Q in iteration t – a contradiction. ∎

Procedure parallelParetoSearch(G, s)
  $L[v]$:= $\emptyset$ for all $v \in V$; $L[s]$:= $\{0^d\}$
  ParetoQueue $Q = \{(s, 0^d)\}$
  while $Q \neq \emptyset$ **do**
    remove any Pareto optimal subset L from Q
    scan the labels in L –– see also Section II
    insert new locally nondominated labels $(v, \vec{\ell})$ into Q
      and remove old locally dominated labels

Fig. 1. Pseudo code for (parallel) multi-objective search.

In the following, we concentrate on the most parallel version of the algorithm that scans *all* Pareto optimal labels in Q in each iteration. We call this algorithm paPaSearch. For an example refer to Appendix A. Somewhat surprisingly, although the paPaSearch does not simply scan one node per iteration as Dijkstra's algorithm, it still has the same bound on the number of iterations. Indeed, we prove a little bit more: For any Pareto optimal label value $\vec{\ell}$ in Q it suffices to scan just *one* queue entry (there might be several entries with the same value of $\vec{\ell}$ for different nodes). We can postpone scanning of the remaining entries with value $\vec{\ell}$ to subsequent iterations. This gives us some flexibility in implementing the queue operations. In particular, this means that numerical inaccuracies when computing floating point labels are unlikely to be a big problem.

Theorem 2: paPaSearch performs at most n iterations.

Proof: We model the computation performed by paPaSearch in an abstract way by defining a dependence graph D whose vertices are all the labels ever scanned by the algorithm and where a dependence arc (a, b) means that label $a = (v_a, \vec{\ell}_a)$ may be scanned before label $b = (v_b, \vec{\ell}_b)$ by some possible run of the algorithm. There are two kinds of such dependencies. Either $\vec{\ell}_a \preceq \vec{\ell}_b$ ((weak) dominance) or an edge between v_a and v_b is relaxed during the execution of the algorithm creating the label b. However, since the latter arcs are a subset of the former arcs, we can concentrate on the dominance arcs. The number of iterations T of paPaSearch is witnessed by a simple path in D with T nodes, i.e., it suffices to show that D does not contain a simple path with more than n nodes. This is true because no node of the input graph G can appear in any simple path through D multiple times: Suppose the contrary were true, i.e., D contains a simple path $(v, \vec{\ell}) \to \cdots \to (v, \vec{\ell}')$. This would imply that $(v, \vec{\ell}) \preceq (v, \vec{\ell}')$. But this cannot happen since the edge relaxation considering $(v, \vec{\ell}')$ would find that $\vec{\ell}'$ is not new or dominated at v, i.e., the label would not be inserted into Q and thus cannot be scanned later – a contradiction. ∎

Implementation Details

The exact way paPaSearch is implemented depends on d and on the model of computation used. We describe a "model-agnostic" approach using well known primitives for parallel computations including collective operations such as prefix sums and task scheduling algorithms such as work stealing. This is then easy to translate to various shared memory models of computation. In this section we provide as much information as possible for a general approach independent of the number of objectives. In Section V we fill in the details for the case $d = 2$. The algorithm maintains data structures for Q, for G, and for the set $L[v]$ of local labels at each node v.

Load balancing using prefix sums: We repeatedly use a well known technique to assign tasks to processors using prefix sums. Consider k tasks with sizes $t_1, \ldots, t_k$ and their prefix sums $T_i := \sum_{j \le i} t_j$ (also define $T_0 := 0$). Note that these sums can be computed in time $\mathcal{O}(k/p + \log p)$ (e.g., [14]).

We assign task i to *work slots* $T_{i-1}..T_i - 1$. By assigning work slots $(i-1)T_k/p + 1..iT_k/p$ to processor i we achieve an equal distribution of work. Some (small) tasks may be exclusively assigned to a single processor this way. Some tasks, in particular those with size exceeding T_k/p will be assigned to several processors. We will call those tasks *split*. Note that each processor works on at most two split tasks. An asymptotically efficient way of actually executing the tasks is to first process the exclusive tasks. Number the split tasks using a prefix sum. Subsequently, process the even numbered tasks and finally the odd numbered ones.[4] Overall, the tasks can be processed in time $\mathcal{O}(T_k/p + \log p)$ plus the time needed for coordinating processors cooperating on split tasks which depends on the way these tasks are parallelized.

1: Pareto queue: The set of Pareto optimal labels L has to be identified and removed from Q.

2: Generating candidates: We can then generate new candidate labels – for each removed label $(u, \vec{\ell})$ and each edge $e = (u, v)$ we generate a candidate label $(v, \vec{\ell} + \vec{c}(e))$. Load balancing for this phase can be done by performing a prefix sum over the degrees of the nodes L. This can be used to assign an equal number of candidate labels to each processor.

3: Grouping candidates: Candidate labels are then grouped by node ID. For example this can be done by sorting, possibly exploiting additional properties of the set of candidates. We then assign an equal number of sorted candidate labels to each processor using another prefix sum computation. Note that the candidates $(v, \vec{\ell})$ concerning the same node v are thus assigned to a consecutively numbered range of processors.

4: Computing Pareto optima among candidates: Next, those candidate labels are removed that are dominated by other candidate labels at the same node. This operation entails an independent computation of Pareto optimal point sets for each node. Some of these computations involve multiple processors, so that we need a parallel algorithm for this problem. Once more, these calculations can be scheduled using prefix sums as long as we can predict the work needed for each task.

5: Merging new and old labels: The surviving candidate labels are now compared with the previously existing labels – again independently for each node, and in parallel if the number of labels for a node is large. Candidates that are dominated by an old label or equal to an old label are dropped. Dominated old labels are removed from the local label set and remembered for later removal from Q. Surviving candidate labels are inserted into the local label sets.

6: Bulk update of Q: After these node-local computations, a bulk update is performed on the global label set Q – surviving label candidates are inserted and newly dominated old labels are removed.

V. A Bicriteria Implementation

In this section we name the two objectives as x and y coordinates. The main observation we exploit here repeatedly is that a 2D-Pareto set can be maintained as a sequence sorted by increasing x-coordinate and decreasing y-coordinate. Since we are analyzing a parallelization of a sequential algorithm (sequential label setting biobjective search), the running time of the parallel algorithm will necessarily not only depend on the input size but also on the number of operations performed by the sequential algorithm. Thus, let N denote the total number of label scans and M the total number of edge relaxations performed by biobjective paPaSearch. Sections V-A and V-B show the following result for parallel execution time in a shared memory parallel model:

Theorem 3: paPaSearch on p processors can run in time

$$\mathcal{O}\left(\frac{(N+M)\log M}{p} + n \log p \log M\right) .$$

Essentially this means logarithmic work per edge relaxation which is the same one would obtain for a sequential label-setting algorithm.[5] Only for instances with little work per processor will we experience a polylogarithmic slowdown compared to Dijkstra's algorithm. Theorem 2 already tells us that we have at most n iterations of the main loop so that it remains to analyze the operations involved in finding and scanning optimal labels from the Pareto queue Q. Sections V-A and V-B explain these operations verbally. Appendix A gives an example. Appendix B summarizes the algorithm using pseudo code. In Section V-C we show how the parallel nature of paPaSearch can be exploited to reduce the work performed by the algorithm proving the following theorem:

Theorem 4: paPaSearch can be implemented to perform work $\mathcal{O}((N+M)\log n)$.

Note that for difficult instances where the number of edge relaxations M is superpolynomial in the size of the graph, this is an asymptotic improvement compared to the complexity of the classical, priority queue based sequential algorithm which is $\Theta((N+M)\log M)$. The graph with exponential output size from the introduction is an example. For simplicity, we restrict the description to sequential algorithms but there is nothing to prevent us from also parallelizing this algorithm.

A. Node Local Operations

4: Computing Pareto optima among candidates: From a set of points lexicographically sorted by increasing x and y-coordinate, we can easily determine the Pareto optimal points by computing prefix minima over the y-coordinates – the optima are those points where new minima of the y-coordinate are encountered.

5: Merging new and old labels: The difficulty here is that there may be a large number of old labels and a small number of new labels so that it would be inefficient to always spend work proportional to the number of old labels. This can be avoided by storing the old labels in a search tree ordered by increasing x coordinate (and, implicitly, by decreasing y

[4] In concrete practical cases one can probably come up with more efficient ways to do this.

[5] Using Fibonacci heaps, Dijkstra's single-objective algorithm performs only constant work per edge relaxation but this does not translate to biobjective search since already checking whether a candidate label is dominated requires search in a search tree data structure.

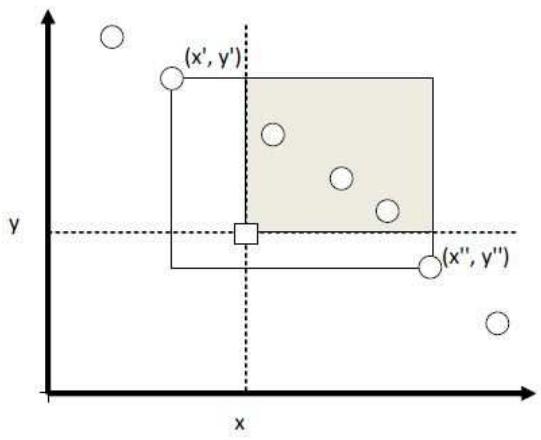


Fig. 2. Sample local label set in 2D. A new label (x, y) is inserted. The predecessors in x-coordinate (x', y') and y-coordinate (x'', y'') are displayed. Any label in the shaded region will be dominated.

coordinate). Our algorithm locates each new label $\vec{\ell} = (x, y)$ in this tree finding the point $\vec{\ell'} = (x', y')$ with maximal x' such that $x' \leq x$ (a predecessor query).[6] If $y \geq y'$, $\vec{\ell}$ is dominated by $\vec{\ell'}$ or equal to it and can be discarded. Otherwise, $\vec{\ell}$ is locally Pareto optimal and may also dominate a range of old labels. This range can be determined by locating $\vec{\ell}$ with respect to y-coordinate. Figure 2 displays a sample case in 2-dimensional cost space. The new label dominates previous labels included in the shaded region, if any.

The result of this computation is a set of new labels to be inserted into the tree together with a set of ranges of old labels to be removed. These ranges can be represented by a range of x values. The corresponding y-values are implicit. The set of x-ranges can be converted into a set of nonoverlapping ranges by exploiting their special structure. Sort the ranges $[x_i, x_i')$ such that the x_i increase. If several ranges have equal x_i, we consider only one with the largest x_i'. Then the x_i' values also form an increasing sequence. Hence, we can prune x_i' to $\min(x_i', x_{i+1})$ without losing coverage of elements to be removed. Overall, we get sorted sequences of bulk update operations to the set of old labels (insertions and nonoverlapping range removals). The pseudo code in the appendix first removes ranges of old labels and then inserts the new labels.

A simple way to perform such bulk updates is to split the sequence of updates and the search tree into p corresponding pieces. Then each processor can apply one subsequence of updates to the corresponding tree. Finally, the trees are concatenated again. This has been proposed for insertion in [8]. Using recursive splitting and concatenation, this can be done in time $\mathcal{O}((k/p + \log p) \log |T|)$ for k operations and a tree of size T. See Appendix B for details. Since the size of the tree is bounded by M, the number of steps is bounded by n, and the total number of updates sums to at most $N + M$, we get a total contribution of $\mathcal{O}\Big(\frac{(N+M)\log M}{p} + n \log p \log M\Big)$ time for bulk updates. Note that we do not necessarily have to unpack ranges of labels to be deleted into individual label removals since splitting and concatenation operations can also be used to remove arbitrarily large ranges in logarithmic time. If we want to avoid the resulting memory management issues, we can also use explicit unpacking together with additional load balancing operations (using prefix sums once more). The additional work implied by unpacking is no problem since it can be charged to the operations originally inserting the elements.

B. The Pareto Queue Q

We simplify a data structure described in [4] (and in some of the papers cited there) to obtain a simple solution for the operation mix needed in our application that yields a good basis for parallelization. We store the elements of Q in the leaves of a balanced binary search tree lexicographically sorted by increasing x, y-coordinate, and node ID, e.g., a red-black tree [12]. The tree is augmented by storing in each interior node v the leaf with minimum y value $v.\min$ occurring in the subtree rooted at v. If there are several such values, the leftmost one is chosen. Algorithm findParetoMinima given in Figure 3 uses this data structure to report all Pareto minima not dominated by parameter u. We ensure the precondition that the procedure is only called on subtrees that actually contain a result. If the min-value from the left subtree dominates the min-value from the right subtree, there will be no results from the right subtree, i.e., only the left subtree has to be searched. Similarly, if u dominates the min-value from the left subtree, only the right subtree has to be searched. Otherwise, both subtrees can be searched in parallel. We only have to be careful not to report values from the right subtree that are dominated by values from the left subtree. This is achieved by passing the min-value $(\check{x}, \check{y})$ from the left subtree as a new bound u to the call for the right subtree. To see that this works, assume that a value (x, y) from the left subtree dominates a value (x', y') from the right subtree. By definition of the data structure we have $\check{y} \leq y \leq y'$ and $\check{x} \leq x'$, i.e., certainly $(\check{x}, \check{y}) \preceq (x', y')$. The case $(\check{x}, \check{y}) = (x', y')$ cannot occur since this would imply $(x, y) \prec (\check{x}, \check{y})$ which contradicts the definition of $(\check{x}, \check{y})$ as the leftmost entry with minimal y-value.

In a task parallel programming language such as Cilk[7] this kind of parallel algorithm is easy to implement and we get parallel execution $\mathcal{O}(W/p + T_\infty)$ where W is the total work performed and T_∞ is the critical path length of the computation (the longest path through the computation DAG) [1]. The critical path length is bounded by the depth of the tree which is logarithmic in $|Q|$. The total work is $\mathcal{O}(\log |Q|)$ per reported result label. Since there are at most N result labels and since $|Q|$ is bounded by M we get total work $\mathcal{O}(N \log M)$ when summing over all iterations of the algorithm.

6: Bulk update: This data structure also supports the bulk-insert and bulk-remove operations required for updating

[6] Border cases can be handled using a sentinel entry $(-\infty, \infty)$.

[7] http://supertech.csail.mit.edu/cilk/

Procedure findParetoMinima(v, u)
  Assert v contains an element w with $u \not\prec w$
  if v is a leaf then report the label stored at v
  elsif $v \rightarrow$ left.min $\prec v \rightarrow$ right.min **then**
    findParetoMinima$(v \rightarrow$ left$, u)$
  elsif $u \prec v \rightarrow$ left.min **then**
    findParetoMinima$(v \rightarrow$ right$, u)$
  else do in parallel
    findParetoMinima$(v \rightarrow$ left$, u)$
    findParetoMinima$(v \rightarrow$ right$, v \rightarrow$ left.min$)$

Fig. 3. Finding all Pareto minima in the subtree rooted at v with value not dominated by u

the queue. Using the same approach as described in Section V-A. A complication is that the elements of Q that have to be removed are reported as ranges within a local label set. Since they may not represent consecutive elements in Q we have to explicitly unpack them (in contrast to the bulk updates for the local label sets). The work needed for handling the elements individually can be charged to their original edge relaxations. Since ranges can be large, we need another step of explicit prefix sum based load balancing however. In order to be able to quantify the amount of work represented by subtrees in the local label sets, we explicitly store subtree sizes in all interior nodes. We can account for the work performed during these updates by charging $\mathcal{O}(\log M)$ cost to the edge relaxations causing the label to be inserted in the first place. In addition, there is a cost of $\mathcal{O}(\log p \log M)$ for each of the iterations analogously to the bulk updates of the node local queues.

C. More Work Efficient Implementation

The basic idea behind the improvements described in this section is to exploit that the search and update operations performed by paPaSearch are done in a batched fashion. We can therefore make use of several variations of the basic idea of *finger search* [11] to reduce complexity – the complexity goes down from $\log|$size of the data structure$|$ to $\log|$distance between subsequent accesses$|$.

The Pareto Queue Q: The work performed by Algorithm findParetoMinima is proportional to the number of search tree nodes touched which is $\mathcal{O}(|L| \log \frac{|Q|}{|L|})$ by a folklore result that can for example be found in [2, Lemma 2]. Using $|Q| \leq M \leq nN$ and summing over all iterations we obtain a bound of

$$\sum_{i=1}^{n} |L_i| \log \frac{nN}{|L_i|} \leq N \log \frac{nN}{N/n} = N \log n^2 = 2N \log n$$

where L_i is the number of elements removed in the i-th iteration of paPaSearch and the estimate follows from the concavity of the logarithm function and the fact that the average number of removed elements is N/n.

Removing the identified optima and dominated points only takes amortized constant time per element (e.g. see [18, Section 7.4]).

Bulk insertion of new queue elements which are already sorted by x coordinate can profit from finger search so that amortized insertion time is only proportional to the distance from the previously inserted element. A similar argument as above then yields $\mathcal{O}(\log n)$ work per inserted element. To leverage finger search for balanced binary search trees, we just need to remember the path leading to the previous element, searching it bottom up to the place where the search tree branches off to the next element.[8]

Node Local Operations: Similar to the Pareto queue operations, bulk operations on the local distance labels can be reduced when the involved labels are already sorted by x-coordinate. This sortedness property can be established by exploiting that the scanned labels can be generated and scanned in x-sorted order. This sorting is only partially destroyed by the edge relaxations, i.e., the relaxations involving a particular edge remain sorted. This ordering can be preserved if we use a stable sorting algorithm for grouping candidates. The grouping operation itself can be done with linear work using bucket sort using one bucket for each edge.[9] Next, each node regenerates the x-ordering of its candidate labels using multiway merging of the buckets for its incoming edges. This can be done with $\mathcal{O}(\log \text{indegree}) \subseteq \mathcal{O}(\log n)$ work per candidate label.

The subsequent operations are performed in a pipelined fashion. For each new candidate label coming out of the merger, we just have to check whether it is a new minimum with respect to the y-coordinate. If not, it is skipped. A surviving candidate $(v, (x, y))$ performs a finger search in $L[v]$ using the previous x-coordinate for finger information. Using a similar argument as above, the overall number of M candidate labels generated throughout the execution of the algorithm is distributed over n iterations and n nodes so that the average finger search distance is $\mathcal{O}(n^2)$ resulting in an amortized search complexity of $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$.

VI. Single Target Search

Sometimes we are only interested in paths to a particular target node t.[10] Then, labels dominated by an entry in $L[t]$ can be safely discarded, an operation called *filtering*. We can also make search goal-directed by introducing admissible node potentials in the same way as in sequential multi-objective search [16]. The sequential goal-directed multi-objective search algorithm (NAMOA*) has been found to be optimal over a class of admissible algorithms when potentials satisfy a consistency property [16]. This means that no algorithm in this class that uses the same potentials, and always returns the complete set of Pareto optimal solutions, can avoid scanning a label scanned

[8] This is likely to be a folklore observation.

[9] We need a folklore trick here if the number of edge relaxations is much smaller than m in a particular iteration: the bucket array is initialized only once before the first iteration. During an iteration we keep a list of nonempty buckets that is updated when an empty bucket gets an entry. After an iteration only the nonempty buckets need to be emptied.

[10] Multiple interchangeable target nodes can be handled by introducing a single new target node t and zero weight edges connecting the original target nodes to t.

by NAMOA*. We describe a parallelization with the same strong property.

The required lower bounds can be computed using d single-objective searches [24] that are hard to parallelize individually but much faster than the main search. We can also parallelize over the d independent searches. Let us have potentials $\vec{h}(v)\ \forall v \in V$, that lower bound the length of any v–t-path. The priority queue will now store extended labels $(v, \vec{\ell}, \vec{f})$, such that $\vec{f} = \vec{\ell} + \vec{h}(v)$ is an *evaluation* of the length of any path that reaches v with length $\vec{\ell}$ and then continues on to t. Dominance in Q is now established over evaluations ($\vec{f}$). A candidate label $(v, \vec{\ell}, \vec{f})$ can be filtered if its evaluation $\vec{f}$ is dominated by an entry in $L[t]$. Theorem 2 can be generalized to show that the number of iterations is bounded by the number of nodes that are ever touched by the search (refer to Appendix C for details).

There are many ways to implement filtering. Here we consider a simple "lazy" approach that achieves the same number of label scans as the sequential algorithm NAMOA* from [16] and still needs only logarithmic work per edge relaxation: Filtering is performed after a label has been removed from Q but before actually scanning it.[11] Since this amounts to merging the Pareto sets L and $L[t]$, we can use a similar parallel algorithm as described in Section V-A. The costs for labels travelling through Q and for filtering them can all be charged to the original edge relaxations and hence do not increase the overall asymptotic work. Only in analyzing the number of steps, we have to be careful how labels in Q that are filtered out affect the total number of steps.

VII. More Objectives

For $d \geq 3$ it is not known how to efficiently implement a Pareto queue. This makes it unattractive to run the fully parallel version of paPaSearch compared to the sequential label-setting algorithm which can use an ordinary priority queue. However, we can use a 2-d Pareto queue using just two of the objectives. This may already suffice to exhibit a significant amount of parallelism. More generally, we can use any two positive linear combinations of all d objectives for defining the values for the 2-d Pareto queue. For increasing parallelism, a good strategy will be to use uncorrelated or even negatively correlated objectives (or positive linear combinations of them).

VIII. Conclusion

We have introduced a parallel multi-objective algorithm as a natural generalization of Dijkstra's single-objective shortest path algorithm. Somewhat surprisingly, the maximally parallel variant paPaSearch is even closer to Dijkstra's algorithm than the sequential variant since it has the same bound on the number of iterations. This is interesting because it means that all the additional work due to multiple objectives is parallelizable. This is somehow the best we can hope for since single-objective search is notoriously difficult to parallelize in the worst case. For certain inputs, e.g., with low diameter and random edge weights, there are some parallelization opportunities using a label setting algorithm [7] and even more so with some controlled amount of label correction [19]. Combining both sources of parallelism (paPaSearch plus [7] or [19]) is interesting for future work and requires a generalized data structure for the work queue.

For the biobjective case, we also give an efficient implementation of the required data structures. Another surprise is that this implementation is asymptotically more efficient than the sequential algorithm since the batched nature of data structure updates supplies additional information for the required search operations.

As paPaSearch is explained here, it might be too complicated to be practical. We still decided to go to this level of sophistication because this yields elegant and easy to interpret asymptotical results. However we believe that a simplified version of the algorithm might be useful in practice. In particular, we may get away with implementations that work sequentially at each single node in the node local operations. The batched nature of all the operations might also help with cache efficiency. On distributed memory machines or shared memory machines with non uniform memory access (NUMA), we might be able to exploit some locality if the graph is partitioned into compact pieces. However, this locality competes with load balancing which works best if nodes are distributed randomly. Hence, successful parallelizations on such machines will be a challenging problem depending also on the structure of the input graph.

Our results raise a number of further interesting open theoretical questions. Can we also handle the maximally parallel case for $d = 3$ or larger, perhaps at the cost of a polylogarithmic slowdown? Can we reduce the critical path length of the algorithm for $d = 2$ by a factor $\log p$ thus matching the execution time of Dijkstra's algorithm with binary heaps? What other applications can we find for our parallel Pareto queue and parallel bulk updates for search trees, perhaps supplying additional functionality like general range updates?

Acknowledgements: We would like to thank Gerth Brodal, Stephan Erb, Moritz Kobitzsch, and Nodari Sitchinava for valuable help with handling the data structures involved. Peter Sanders is partially supported by DFG projects SA 933/5-2 and SA 933/3-3. Lawrence Mandow is supported by Consejería de Innovación, Ciencia y Empresa, Junta de Andalucía (España), P07-TIC-03018, and Grant TIN2009-14179 (Spanish Government, Plan Nacional de I+D+i).

[11] In practice, one would probably filter earlier and exploit that one does not always have to filter against all labels in $L[t]$.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, pages 356–368, Santa Fe, NM, 20–22 November, 1994. IEEE, Los Alamitos, CA.

[2] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(n \log n)$. *Algorithmica*, 38(2):377–395, 2003.

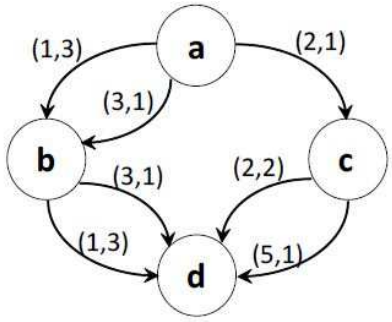
[3] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *J. of Par. and Distr. Comp.*, 49(1):4–21, 25 Feb. 1998.

[4] G. S. Brodal and K. Tsakalidis. Dynamic planar range maxima queries. In *ICALP*, volume 6755 of *LNCS*, pages 256–267. Springer, 2011.
[5] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.
[6] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *23rd MFCS*, number 1450 in LNCS, pages 722–731. Springer, 1998.
[7] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *23rd Symposium on Mathematical Foundations of Computer Science*, number 1450 in LNCS, pages 722–731, Brno, Czech Republic, 1998. Springer.
[8] L. Frias and J. Singler. Parallelization of bulk operations for STL dictionaries. In *Euro-Par Workshops*, volume 4854 of *LNCS*, pages 49–58. Springer, 2007.
[9] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman and Company, New York, 1979.
[10] 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.
[11] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *9th ACM Symposium on Theory of Computing*, pages 49–60. ACM, 1977.
[12] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th IEEE FOCS*, pages 8–21, 1978.
[13] P. Hansen. Bicriterion path problems. In *Multiple Criteria Decision Making Theory and Application*, volume 177 of *LNEMS*, pages 109–127. Springer, 1980.
[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
[15] E. Machuca, L. Mandow, J. L. Pérez de la Cruz, and A. Ruiz-Sepúlveda. A comparison of heuristic best-first algorithms for bicriterion shortest path problems. *European Journal of Operational Research*, 217:44–53, 2012.
[16] L. Mandow and J. L. Pérez de la Cruz. Multiobjective A* Search with Consistent Heuristics. *Journal of the ACM*, 57(5):27:1–25, 2010.
[17] E. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.
[18] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.
[19] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
[20] M. Müller-Hannemann and K. Weihe. On the cardinality of the Pareto set in bicriteria shortest path problems. *Annals OR*, 147(1):269–286, 2006.
[21] J. M. Paixão and J. L. Santos. Labelling methods for the general case of the multi-objective shortest path problem – a computational study. Technical Report DMUC. 07-42, Centro de Matemática da Universidade de Coimbra, 2007.
[22] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, 2009.
[23] A. J. V. Skriver. A classification of bicriterion shortest path (BSP) algorithms. *Asia Pacific Journal Of Operational Research*, 17(2):199–212, 2000.
[24] C. T. Tung and K. L. Chew. A multicriteria Pareto-optimal path algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

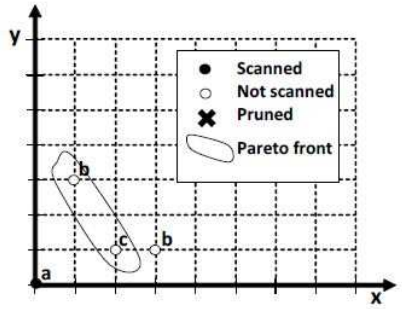
APPENDIX

A. Example

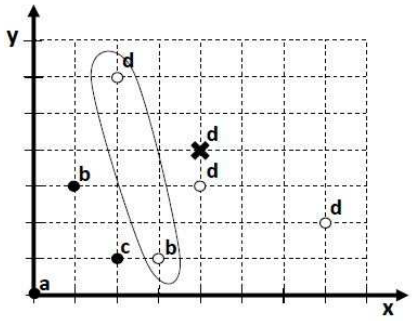
A trace of paPaSearch over a simple graph with two objectives is shown in Figure 4. Figure 4(a) displays a sample graph with four nodes. The start node is a. At the first iteration, the only unscanned label is $(a, (0, 0))$. The sets of scanned, unscanned, and pruned labels are shown for subsequent iterations in figures 4(b), 4(c), and 4(d). At iteration 2 there are three (unscanned) labels in Q, but label $(b, (3, 1))$ is dominated. Therefore, only the other two are scanned in parallel. At iteration 3 there are four labels in Q, but only two are nondominated. Note that four labels were found to node d. However, one is dominated and was pruned from the local label set. During iteration 3 two more labels will be found to node d. One is dominated and pruned, while the other



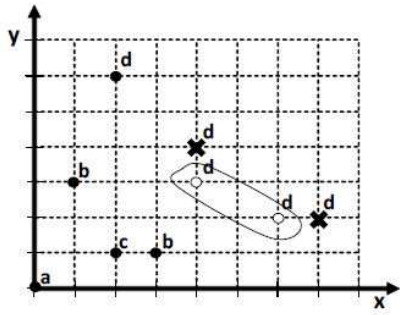
(a) Sample graph



(b) Cost space at iteration 2



(c) Cost space at iteration 3



(d) Cost space at iteration 4

Fig. 4. Sample graph and trace of paPaSearch in cost space.

dominates and prunes an old label in the local label set. After the two remaining labels are scanned in parallel at iteration 4, the queue becomes empty and the algorithm terminates.

In this example, the algorithm performs exactly four iterations (the same as the number of nodes), scanning a total of seven labels.

B. More Details for the 2D Algorithm

Figure 6 gives pseudo code for the biobjective implementation of paPaSearch using high level code that is easy to implement as a data parallel program using standard techniques. In particular, prefix sums are used in several places for distributing work to processors. Figure 5 gives exemplary code for bulk removal of a sequence of ranges from a balanced search tree. The other bulk update operations can be implemented analogously. For a description of the split and concat operations used there refer to [18]. The pseudo code assumes that we can remove an entire range from a search tree in logarithmic time. This can for example be achieved using split and concat once again – appending the cut out pieces to an additional search tree acting as free list. Note that we use time $\mathcal{O}(\log |T|)$ for splitting, concatenation, and individual operations and have a recursion depth of $\log p$ yielding a time bound of $\mathcal{O}((k/p + \log p) \log |T|)$.

C. Number of Steps of Goal Directed Search

Theorem 5: When consistent potentials are considered, goal directed single target paPaSearch as described in Section VI performs at most n' iterations where n' is the number of nodes touched by the search (i.e., nodes for which at least one label enters the queue).

Proof: Let us assume potentials $\vec{h}(v)\ \forall v \in V$, that lower bound the length of any v–t-path. Let us further assume that our potentials satisfy the consistency property [16], i.e. for all pairs of nodes v, w in the graph, and for all nondominated paths between them $P = \langle v, \ldots, w \rangle$ with cost $\vec{c}(P)$, the following condition holds,

$$\vec{h}(v) \preceq \vec{c}(P) + \vec{h}(w)$$

or, the equivalent *monotonicity* property holds: for all arcs $(v, w) \in E$ the following condition holds,

$$\vec{h}(v) \preceq \vec{c}((v, w)) + \vec{h}(w)$$

Consider a similar dependence graph D as in the proof of Theorem 2. Vertices are extended labels in the Pareto optimal subsets L removed from Q at each iteration (which may still be filtered). A dependence arc (a, b) means that $(v_a, \vec{\ell}_a, \vec{f}_a)$ may be scanned before label $(v_b, \vec{\ell}_b, \vec{f}_b)$ by some possible run of the algorithm, and entails that $\vec{f}_a \preceq \vec{f}_b$.

Since D does not contain vertices for untouched nodes, it suffices to show once more that there cannot be simple paths in D that contain a node of G multiple times. So suppose there is a simple path $(v, \vec{\ell}, \vec{f}) \to \cdots \to (v, \vec{\ell}', \vec{f}')$ in D. Notice that $\vec{f} \preceq \vec{f}'$ if and only if $\vec{\ell} \preceq \vec{\ell}'$, since we only consider a single potential for each node. However, we cannot have $\vec{\ell} \preceq \vec{\ell}'$ since then $\vec{\ell}'$ would be pruned from $L[v]$ and thus never enter Q. ■

```
Procedure bulkRangeRemove(T, ⟨I_1, ..., I_k⟩, i..j)
    if k = 1 or i = j then processor i does all the work                    -- time O(k log |T|)
    else
        (T_1, T_2):= T.splitAt(I_{⌊k/2⌋+1})                                 -- by processor i in time O(log |T|)
        dopar
            bulkRangeRemove(T_1, ⟨I_1, ..., I_{⌊k/2⌋}⟩, i.. ⌊(i+j)/2⌋)
            bulkRangeRemove(T_2, ⟨[I_{⌊k/2⌋+1}, ..., I_k⟩, ⌊(i+j)/2⌋ + 1..j)
        T:= concat(T_1, T_2)                                                -- by processor i in time O(log |T|)
```

Fig. 5. Pseudo code for bulk removal of k ranges from a binary search tree T using processors $i..j$.

```
Procedure parallelParetoSearch2D(G, s)
    L[v]:= ∅ for all v ∈ V;   L[s]:= {0^d}
    ParetoQueue Q = {(s, 0^d)}                                              -- (1)
    while Q ≠ ∅ do
        L:= findParetoMinima(Q.root, ∞)-- see Figure 3                      -- (1)
        Q:= Q \ L                                                           -- bulk removal using splitting (6)
        L_c:= ⟨(v, ℓ + c⃗((u,v))) : (u, ℓ) ∈ L ∧ (u, v) ∈ E⟩                 -- generate candidate labels, PS, (2)
        foreach L_c(v):= ⟨ℓ : (v, ℓ) ∈ L_c⟩ dopar   -- sort by v, one task for each node v (possibly parallel), PS (3)
            sort the (x, y) ∈ L_c(v) lexicographically by x and y            -- (4)
            L_new:= the labels (x, y) in L_c(v) with prefix minimal y        -- (4)
            foreach (x, y) ∈ L_new dopar                                     -- (5)
                (x', y'):= L[v].x-predecessor(x)
                if y' > y then                                               -- label (x, y) survives
                    remember (x, y) for insertion into L[v]
                    if x' = x then
                        init:= (x', y')
                    else
                        init:= L[v].successor(x', y')
                    (x'', y''):= L[v].y-predecessor(y)
                    if y'' = y then
                        last:= (x'', y'')
                    else
                        last:= L[v].predecessor(x'', y'')
                    if init.x ≤ last.x then
                        remember range [init, last]
            make deletion ranges nonoverlapping                              -- see text
            remove deletion ranges from L[v]                                 -- bulk removal
            insert new labels into L[v]                                      -- bulk insertion
        L_+:= {(v, ℓ) : ℓ was newly inserted into L[v]}                       -- explicitly compute all removed labels, PS
        L_-:= {(v, ℓ) : ℓ was in a deletion range for L_c(v)}
        Q:= Q ∪ L_+ \ L_-                                                    -- bulk updates (6)
```

Fig. 6. Pseudo code for (parallel) multi-objective search. A comment "PS" means that prefix sums are used for load balancing here. The numbers in brackets refer to the step numbers used in Section IV and V