Alcom-FT Technical Report Series
ALCOMFT-TR-01-95

Parallel Shortest Path for Arbitrary Graphs*

U. Meyer and P. Sanders

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany.
{umeyer,sanders}@mpi-sb.mpg.de.
http://www.mpi-sb.mpg.de/{~umeyer,~sanders}

Abstract. In spite of intensive research, no work-efficient parallel algorithm for the single source shortest path problem is known which works in sublinear time for arbitrary directed graphs with non-negative edge weights. We present an algorithm that improves this situation for graphs where the ratio d_c/Δ between the maximum weight of a shortest path d_c and a "safe step width" Δ is not too large. We show how such a step width can be found efficiently and give several graph classes which meet the above condition, such that our parallel shortest path algorithm runs in sublinear time and uses linear work. The new algorithm is even faster than a previous one which only works for random graphs with random edge weights [10]. On those graphs our new approach is faster by a factor of $\Theta(\log n/\log\log n)$ and achieves an expected time bound of $\mathcal{O}(\log^2 n)$ using linear work.

1 Introduction

The *single source shortest path problem* (SSSP) is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [1]. Let $G = (V, E)$ be a directed graph, $|V| = n$, $|E| = m$, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real-valued *weight* to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the weight of a minimum-weight ("shortest") path from s to v, denoted by *dist(v)*; the weight of a path is the sum of the weights of its edges.

The theoretically most efficient sequential algorithm on directed graphs with non-negative edge weights is Dijkstra's algorithm [5]. Using Fibonacci heaps its running time is given by $\mathcal{O}(n \log n + m)$. Dijkstra's algorithm maintains a partition of V into *settled*, *queued* and *unreached* nodes and for each node v a *tentative distance* $\mathrm{tent}(v)$; $\mathrm{tent}(v)$ is always the weight of some path from s to v and hence an upper bound on $\mathrm{dist}(v)$. For unreached nodes, $\mathrm{tent}(v) = \infty$. Initially, s is queued, $\mathrm{tent}(s) = 0$, and all other nodes are unreached. In each iteration, the queued node v with smallest tentative distance is selected and declared settled and all edges (v, w) are relaxed, i.e., $\mathrm{tent}(w)$ is set to

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

$\min\{\text{tent}(w), \text{tent}(v) + c(v,w)\}$. If w was unreached, it is now queued. It is well known that $\text{tent}(v) = \text{dist}(v)$, when v is selected from the queue.

The only known $\mathcal{O}(n \log n + m)$ work parallel SSSP approach for arbitrary directed graphs based on Dijkstra's algorithm uses parallel relaxation of the edges leaving a single node [7]. It has running time $\mathcal{O}(n \log n)$ on a PRAM[1]. All existing algorithms with sublinear execution time require $\Omega(n \log n + m)$ work (e.g., $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3 (\log \log n / \log n)^{1/3})$ work [8]). Some less inefficient algorithms are known for planar digraphs [15] or graphs with separator decomposition [3].

Higher parallelism than in Dijkstra's approach can be obtained by a version of the Bellman-Ford algorithm [1] which considers all queued nodes with their outgoing edges *in parallel*. However, it may remove nodes v from the queue for which $\text{dist}(v) < \text{tent}(v)$ and hence may have to *reinsert* those nodes until they are finally settled. Reinsertions lead to additional overhead since their outgoing nodes may have to be *rerelaxed*.

The present paper is based on the Δ-stepping algorithm of [10] which is a generalization of Dijkstra and Bellman-Ford: Tentative distances are kept in an array B of *buckets* such that $B[i]$ stores the unordered set $\{v \in V : v \text{ is queued and } \text{tent}(v) \in [i\Delta, (i+1)\Delta)\}$. In each *phase*, the algorithm removes all nodes from the first nonempty bucket and relaxes all *light* edges ($c(e) \leq \Delta$) of these nodes. This may cause *reinsertions* into the current bucket. For the remaining *heavy* edges, it is sufficient to relax them once and for all when a bucket finally remains empty (see Figure 1). The parameter Δ should be small enough to keep the number of reinsertions small yet large enough to exhibit a useful amount of parallelism.

1.1 Overview and Summary of New Results

The simple parallelization of the Δ-stepping in [10] relies on the particular properties of random graphs with random edge weights thus severely limiting its usage. In Section 2, we introduce a parallel Δ-stepping algorithm which works for arbitrary graphs in time $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta \log n)$ and work $\mathcal{O}(m + n_{\Delta+})$ whp[2]. The parameters which depend on the graph class and the step width are explained in Section 1.2. A further acceleration is achieved in Section 3 by actively introducing *shortcut edges* into the graph thereby reducing the number of times each bucket is emptied to at most two, i.e., the fastest efficient parallel execution time is now $\mathcal{O}((l_\Delta + d_c/\Delta) \log n)$ while performing $\mathcal{O}(m + n'_{\Delta+})$ work whp. In Section 4 it is explained how a good value for the step width Δ (which limits $n'_{\Delta+}$ to $\mathcal{O}(m)$) can be determined efficiently and in parallel. Many of the PRAM results can be adapted to distributed memory machines using techniques described in Section 5. Finally, in Section 6 we summarize the results and apply them on different

[1] We use the *arbitrary CRCW PRAM* model (concurrent read concurrent write parallel random access machine) [9] which specifies that an adversary can choose which access out of a set of conflicting write accesses is successful.

[2] A result holds *with high probability* (*whp*) in the sense that the respective bound is met with probability at least $1 - n^{-\beta}$ for any constant $\beta > 0$.

```
for each v ∈ V do tent(v) := ∞
relax(s, 0);                                          (* Source node at distance 0 *)
while ¬isEmpty(B) do                                  (* Some queued nodes left *)
    i := min{j > i : B[j] ≠ ∅}                        (* Smallest nonempty bucket *)
    R := ∅                                            (* No nodes deleted for bucket B[i] yet *)
    while B[i] ≠ ∅ do                                 (* New phase *)
        Req := findRequests(B[i], light)              (* This may reinsert nodes *)
        R := R ∪ B[i]; B[i] := ∅                      (* Remember deleted nodes *)
        relaxRequests(Req)
    Req := findRequests(R, heavy)                     (* This may reinsert nodes *)
    relaxRequests(Req)

Function findRequests(V', kind : {light, heavy}) : set of Request
    return {(w, tent(v) + c(v, w)) : v ∈ V' ∧ (v, w) ∈ E_kind)}

Procedure relaxRequests(Req) for each (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                                 (* Shorter path to w? *)
    if x < tent(w) then                               (* Yes: decrease-key or insert *)
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}        (* Remove if present *)
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {w}
        tent(w) := x
```

Fig. 1. Sequential Δ-stepping.

graph classes. Although our new algorithm is more general than the specialized previous algorithm [10], it turns out to be a factor of $\Theta(\log n/ \log\log n)$ faster on random graphs. It has execution time $\mathcal{O}(\log^2 n)$ using linear work.

1.2 Notation and Basic Facts

We have already used d_c as an abbreviation for the maximum weight of a shortest path, i.e., $d_c := \max\{\text{dist}(v) : \text{dist}(v) < \infty\}$. Call an edge disjoint path with weight at most Δ a *Δ-path*. Let C_Δ denote the set of all node pairs $\langle u, v\rangle$ connected by some Δ-path $(u, \ldots, v)$ and let $n_\Delta := |C_\Delta|$. Similarly, define $C_{\Delta+}$ as the set of triples $\langle u, v', v\rangle$ such that $\langle u, v'\rangle \in C_\Delta$ and (v', v) is a light edge and let $n_{\Delta+} := |C_{\Delta+}|$. Let n'_Δ ($n'_{\Delta+}$) denote the number of *simple* Δ-paths (plus a light edge). To simplify notation, we exclude very extreme graphs and assume $n = \mathcal{O}(m)$, $n_\Delta = \mathcal{O}(n_{\Delta+})$ and $n'_\Delta = \mathcal{O}(n'_{\Delta+})$. The *maximum Δ-distance* l_Δ is defined to just exceed the number of edges needed to connect any pair $\langle u, v\rangle \in C_\Delta$ by a path of minimum weight, i.e.,

$$l_\Delta = 1 + \max_{\langle u,v\rangle \in C_\Delta} \min\{|A| : A = (u, \ldots, v) \text{ is a minimum-weight } \Delta\text{-path}\} \ .$$

Similarly, let l'_Δ denote the number of edges in the longest simple Δ-path.

The graph theoretic results from [10] are relatively easy to generalize to see that the number of phases performed by Δ-stepping is bounded by $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta)$ and that the number of reinsertions (rerelaxations) is at most n_Δ ($n_{\Delta+}$). For details refer to the full paper [11] which is available electronically.

2 Parallelization

In this section we develop a first parallelization of Δ-stepping which works for arbitrary graphs and prove the following bound:

Theorem 1. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum path weight d_c, maximum Δ-distance l_Δ and $n_{\Delta+}$ defined as in Section 1.2 can be solved on a CRCW PRAM in time $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta \log n)$ and work $\mathcal{O}(m + n_{\Delta+})$ whp.*

Initialization, loop control, deleting nodes and generating a set 'Req' of node-distance pairs to be relaxed (we call these *requests*) are easy to do in parallel if the nodes are randomly assigned to PUs and if a global array stores the assignment.

The most difficult part is to schedule PUs for actually performing the requests: several relaxations can occur for one node in a phase, and the number of such conflicting relaxations can vary arbitrarily and in an unpredictable way. On CRCW-PRAMs, we can do the PU scheduling efficiently by grouping the requests according to the addressed nodes using the following lemma:

Lemma 1. Semi-sorting *k records with integer keys, i.e., permuting them into an array of size k such that all records with equal key form a consecutive block, can be performed in time $\mathcal{O}(k/p + \log n)$ using p PUs of a CRCW-PRAM whp.*

Proof. First find a perfect hash function $h : V \rightarrow 1..ck$ for an appropriate constant c. Using the algorithm of Bast and Hagerup [2] this can be done in time $\mathcal{O}(k/p + \log n)$ (and even faster) whp. Subsequently, we apply a fast, work efficient sorting algorithm for small integer keys such as the one by Rajasekaran and Reif [13] to sort by the hash values.

Once the set of requests 'Req' is grouped by receiving nodes w, we can use prefix sums to schedule $\lfloor p\,|\mathrm{Req}(w)|\,/\,|\mathrm{Req}|\rfloor$ PUs for blocks of size at least $|\mathrm{Req}|\,/p$, and to assign smaller groups with a total of up to $|\mathrm{Req}|\,/p$ requests to individual PUs. The PUs concerned with a group collectively find a request with minimum distance in time $\mathcal{O}(|\mathrm{Req}|\,/p + \log p)$ and then relax it in constant time.

Summing the work and time for all $l_\Delta d_c/\Delta$ phases yields the desired bound.

3 Finding Shortcuts

In the analysis of the number of phases for our algorithms we bounded the maximum number of iterations, l_Δ, that are required until the current bucket under consideration remains finally empty. It was already noticed in [6] that only one iteration per bucket is needed if the bucket width is smaller than any edge weight. No reinsertions occur in that case but in the presence of very small edge weights, the number of buckets, d_c/Δ, might become very large due to the small Δ. However, l_Δ can be reduced to 2 by explicitly introducing a *shortcut edge* (u, v) for each node pair connected by a Δ-path.

What interests us here is how to find these edges in parallel and how the search itself can be performed in a load balanced way. Although, we do not know a general algorithm doing that using $\mathcal{O}(m + n_{\Delta+})$ work, we can solve the problem if the number of simple Δ-paths is not too large. More precisely, the remainder of this section is devoted to establishing the following Theorem:

Theorem 2. *There is an algorithm which inserts an edge* (u, v) *with weight* $c(u, v) = \mathrm{dist}(u, v)$ *for each shortest path* $(u, \ldots, v)$ *with* $\mathrm{dist}(u, v) \leq \Delta$ *using* $\mathcal{O}(l'_\Delta \log n)$ *time and* $\mathcal{O}(m + n'_{\Delta+})$ *work on a CRCW PRAM whp. (Where* $\mathrm{dist}(u, v)$ *denotes the weight of a shortest path from* u *to* v*.)*

Applying the results from Section 2 we get:

Corollary 1. *The single source shortest path problem for directed graphs with* n *nodes,* m *edges, maximum path weight* d_c *and* $n'_{\Delta+}$, l'_Δ *as defined in Section 1.2 can be solved on a CRCW PRAM in time* $\mathcal{O}((l'_\Delta + \frac{d_c}{\Delta}) \log n)$ *and work* $\mathcal{O}(m + n'_{\Delta+})$ *whp.*

Figure 2 outlines a routine which finds shortcuts by applying a variant of the Bellman-Ford algorithm to all nodes in parallel. It solves an all-to-all shortest path problem constrained to Δ-paths. The shortest connections found so far are kept in a hash table of size $\mathcal{O}(n'_{\Delta+})$ (we can use dynamic hashing if we do not know a good bound for $n'_{\Delta+}$). This table plays a role analogous to that of $\mathrm{tent}(\cdot)$ in the main routine of Δ-stepping. The set Q stores *active* connections, i.e., triples (u, v, y) where y is the weight of a shortest known path from u to v and where paths $(u, \ldots, v, w)$ have not yet been considered as possible shortest connections from u to w with weight $y + c(v, w)$. In iteration i of the main loop, the shortest connections using i edges are computed and are then used to update 'found'. Applying similar techniques as before, this routine can be implemented to run in $\mathcal{O}(l'_\Delta \log n)$ parallel time using $\mathcal{O}(m + n'_{\Delta+})$ work: We need l'_Δ iterations each of which takes time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|Q'|)$ whp. The overall work bound holds since for each simple Δ-path $(u, \ldots, v)$, $\langle u, v\rangle$ can be a member of Q only once. Hence, $\sum_i |Q| \leq n + n'_\Delta$ and $\sum_i |Q'| \leq n + n'_{\Delta+}$.

```
Function findShortcuts(Δ) : set of weighted edge
    found : HashArray[V × V]                         (* return ∞ for undefined entries *)
    Q := {(u, u, 0) : u ∈ V}                          (* (start, destination, weight) *)
    Q' : MultiSet
    while Q ≠ ∅ do
        Q' := ∅
        for each (u, v, x) ∈ Q dopar
            for each light edge (v, w) ∈ E dopar
                Q' := Q' ∪ {(u, w, x + c(v, w))}
        semi-sort Q' by common start and destination node
        Q := {(u, v, x) : x = min{y : (u, v, y) ∈ Q'}}
        Q := {(u, v, x) ∈ Q : x ≤ Δ ∧ x < found[(u, v)]}
        for each (u, v, x) ∈ Q dopar found[(u, v)] := x
    return {(u, v, x) : found[(u, v)] < ∞}
```

Fig. 2. CRCW-PRAM routine for finding shortcut edges

4 Determining Δ

In the case of arbitrary edge weights it is necessary to find a step width Δ which is large enough to allow for sufficient parallelism and small enough to keep the algorithm work-efficient. Although we expect that application specific heuristics can often give us a good guess for Δ relatively easily, for a theoretically satisfying result we would like to be able to find a good Δ systematically.

We now explain how this can be done if the adjacency lists have been preprocessed to be *partially sorted*: Let $\Delta_0 := \min_{e \in E} c(e)$ and assume[3] that $\Delta_0 > 0$. The adjacency lists are organized into *blocks* of edges with weight $2^j \Delta_0 \leq c(e) < 2^{j+1} \Delta_0$ for some integer j. Blocks with smaller edges precede blocks with larger edges.[4]

Theorem 3. *Let n'_Δ, $n'_{\Delta+}$ and l'_Δ be defined as in Section 1.2 and consider an input with partially sorted adjacency lists. For any constant α, there is an algorithm which identifies a step width Δ, such that $n'_{\Delta+} \leq \alpha m$ and $n'_{2\Delta+} > \alpha m$, and which can be implemented to run in $\mathcal{O}((l'_\Delta + \log \frac{\Delta}{\Delta_0}) \log n)$ time using $\mathcal{O}(m)$ work whp.*

The basic idea is to reuse the procedure findShortcuts(Δ) of Figure 2 but to divide the computation into *rounds*. In round i, $0 \leq i \leq \log \frac{\max_{e \in E} c(e)}{\Delta_0}$), we set $\Delta_{\text{cur}} = 2^i \Delta_0$ and find all connections (u, v, x) with $\Delta_{\text{cur}} \leq x < 2\Delta_{\text{cur}}$. In order to remain work efficient, a number of additional measures are necessary however. We now outline the changes compared to the routine 'findShortcuts' from Figure 2. Most importantly, we have a bucketed *todo-list* T. $T[i]$ stores entries (u, v, x, b) where (u, v, x) stands for a connection from u to v with weight x, and b points to the first block in the adjacency list of v which may contain edges (v, w) with $2^i \Delta_0 \leq x + c(v, w) < 2 \cdot 2^i \Delta_0$. (Note that the number of buckets may be arbitrarily large. In this case, we store the buckets in a dynamic hash table and only initialize those buckets which actually store elements.)

At the beginning of round i, for each entry (u, v, x, b) of $T[i]$, the adjacency list of v is scanned beginning at block b until a block is encountered which cannot produce any candidate connections for bucket i. A new entry of the todo list is produced for the first bucket $k > i$ for which it can produce candidate connections. The candidate connections found are used to initialize Q'.

Both this initialization step and the iteration on Q can produce candidate connections whose weights reach into bucket $i+1$. After removing duplicates and longer connections than found before, we therefore split the remaining candidates into the new content of Q and a set Q_{next} storing connections with weight in bucket $i + 1$.

At the end of round i, when Q finally remains empty, we create new entries in the todo-lists for all connections newly encountered in round i. In order to do that, we keep track of all new entries into 'found' using two sets S and S_{next} for connections with weights in bucket i and $i + 1$, respectively. Q_{next} and S_{next} are used to initialize Q and S in the next round respectively.

---

[3] This assumption can be removed.

[4] This preprocessing is trivially parallelizable on a node-by-node basis, we get a good parallel preprocessing algorithm for the case $p = \mathcal{O}(n/d)$ if d is the maximum out-degree of a node.

The total number of connection-edge pairs considered is monitored so that the whole procedure can be stopped as soon as it is noticed that this figure exceeds αm. At this time, the entries of 'found' constitute at least all simple $(\Delta_{\mathrm{cur}}/2)$-paths. Thus, taking $\Delta := \Delta_{\mathrm{cur}}/2$ as the final step width, it is guaranteed that the number of reinsertions and rerelaxations in a subsequent application of the Δ-stepping will be bounded by $\mathcal{O}(m)$. On the other hand, $n'_{2\Delta+} > \alpha m$.

Using an analogous analysis as for the function 'findShortcuts' it turns out that the search for Δ can be implemented to run in $\mathcal{O}((l'_\Delta + \log \frac{\Delta}{\Delta_0}) \log n)$ time using $\mathcal{O}(m)$ work where l'_Δ denotes the number of edges in the longest simple Δ-path.

5 Adaptation to Distributed Memory Machines

In this section we consider the following distributed memory model: There are p processing units (PUs) numbered 0 through $p-1$ which are connected by a communication network. Let $T_{\mathtt{routing}}(k)$ denote the time required to route k constant size messages per PU to random destinations. Let $T_{\mathtt{coll}}(k)$ bound the time to perform a (possibly segmented) reduction or broadcast involving a message of length k and assume that $T_{\mathtt{coll}}(x)+T_{\mathtt{coll}}(y) \leq T_{\mathtt{coll}}(1)+T_{\mathtt{coll}}(x+y)$, i.e., concentrating message length does not decrease execution time. Note, that on powerful interconnection networks like multiported hypercubes we can achieve a time $\mathcal{O}(\log p + k)$ whp for $T_{\mathtt{routing}}(k)$ and $T_{\mathtt{coll}}(k)$.

So far it is unknown how to efficiently implement the linear work semi-sorting procedure for load-balancing on distributed memory[5]. However, if shortcuts are present we now explain how this problem can be circumvented. We also assume that the nodes can be randomly assigned to PUs using a constant time hash function[6] $\mathrm{ind}(\cdot)$ and that we know $\mathrm{indegree}(v)$ when looking at an edge (u, v).

Theorem 4. *Given a directed graph G with n nodes, m edges, maximum path weight d_c and $n_{\Delta+}$, l_Δ as defined in Section 1.2. Under the assumptions given above, the single source shortest path problem can be solved in time*

$$\mathcal{O}\left(\overline{m} + T_{\mathtt{routing}}(\overline{m}) + T_{\mathtt{coll}}(\overline{m}) + \frac{d_c}{\Delta}(T_{\mathtt{coll}}(1) + T_{\mathtt{routing}}(1))\right)$$

on a distributed memory machine with p PUs for $\overline{m} = \frac{m+n_{\Delta+}}{p}$ and any given source node s whp.

We first simplify the search algorithm to exploit the fact that in the presence of shortcuts, classifying edges as light or heavy is no longer important for the

[5] The preprocessing can be done (somewhat inefficiently) by implementing semi-sorting using ordinary sorting or using a slower yet work efficient algorithm requiring $\mathcal{O}(T_{\mathtt{routing}}(n^\epsilon))$ time for any positive constant ϵ. Both alternatives yield a work-efficient algorithm for powerful interconnection networks if the preprocessing overhead can be amortized over sufficiently many source nodes.

[6] This is a common assumption, e.g., in efficient PRAM simulation algorithms.

shortest path search itself. By explicitly treating intra-bucket edges (source and target reside in the same bucket) first, each edge is relaxed at most once: After buckets 0 through $i-1$ have been emptied, a single relaxation pass through the edges reaching from $B[i]$ into $B[i]$ suffices to settle all nodes now in $B[i]$. After that, $B[i]$ can be emptied by relaxing all edges reaching out of $B[i]$ once.

The two most difficult parts are (1) generating the set of requests, i.e. identifying the set of edges that are to be relaxed and (2) assigning the requests to their nodes and scheduling the PUs for performing the relaxations. We start with (1):

In a distributed memory setting we cannot dynamically schedule outgoing edges between the PUs in the same way as we did for PRAMs. Scanning adjacency lists to generate requests is therefore load balanced using a static assignment of edges to PUs: An adjacency list of size outdegree(v) is collectively handled by an *out-group* of PUs. Out-groups are selected as follows: W.l.o.g., assume that p is a power of two minus one and the PUs are logically arranged as a complete binary tree. If outdegree$(v) > p$ then all PUs participate in v's out-group. Otherwise, a subtree rooted at a random PU is chosen which is just large enough to accommodate one edge per PU, i.e., it contains $2^{\lceil \log(\text{outdegree}(v)+1)\rceil} - 1$ nodes. Requests for a bucket can now be generated by first sending the tentative distance of the nodes in $B[i]$ to the roots of out-groups responsible for them. (We will later see where this information comes from.) Then, the PUs pass all the node-distance pairs they have received down the tree in a pipelined fashion and do the same for the distances of the nodes received from above.

Now consider a fixed leaf PU j for a fixed iteration of the algorithm. (Since interior tree-nodes pass all their work downwards, interior PUs have no more work to do than a leaf node.) Let $X_i := 1$ if PU j is part of the out-group of a node i expanded in this iteration and $X_i = 0$ otherwise. We have $\mathbf{P}[X_i = 1] = 2^{-h(i)}$ if the root of the out-group of node i is $h(i)$ levels away from the root of the PU-tree. The total number of nodes PU j has to work on is $Y := \sum_{i=1}^{k} X_i$ if k is the number of nodes expanded in the current iteration and $\mathbf{E}[Y] = \sum_i 2^{-h(i)}$. By definition of the size of subtrees, we get $\mathbf{E}[Y] = \mathcal{O}(K/p)$ if K is the total number of edges leaving nodes expanded in this iteration. Using a Chernoff bound with nonuniform probabilities [12, Theorem 4.1], it is now easy to see that $Y = \mathcal{O}(K/p + \log n)$ whp. Since the communication pattern is just a slightly generalized form of a broadcast, distributing the tentative distances can be done in time $\mathcal{O}(T_{\texttt{coll}}(K/p + \log n))$ whp. Summing over all iterations we get time $\mathcal{O}(T_{\texttt{coll}}(m/p + \log n) + T_{\texttt{coll}}(1)d_c/\Delta)$. Generating the actual request values is then possible using local computations only.

Now we tackle problem (2): how to assign the requests to nodes and schedule PUs for performing the relaxations. The idea for arbitrary graphs is to postpone the relaxation of an edge until the latest possible moment – just before the bucket of the target node is emptied. Since edges are relaxed only once (recall that we assume the presence of short-cuts), it pays to allocate an *in-group* of size $2^{\lceil \log(\text{indegree}(v)+1)\rceil} - 1$ for node v analogously to the way out-groups are allocated. Each PU maintains an additional bucket structure B_q for the nodes for which it is part of the in-group. Requests are routed to a preassigned position in the in-group, but this information is only used to place the node into B_q. So, after iteration $i-1$ is computed, the content of $B[i]$ is not yet known. Rather, we first have to find $B[i] = \bigcup B_q[i]$. This can be done locally for each in-group using a pipelined tree operation which is the converse of the operation used for broadcasting in the out-groups. (Each PU maintains a hash table of nodes already passed up the tree.) Then, the result is broadcast to all PUs in the in-groups so that from now on, redundant entries of nodes in buckets beyond $B[i]$ can be deleted. Also, edges which have not received a request yet are marked as superfluous. Requests ending up there in later iterations will simply be discarded. Finally, the actual global minima are computed using another pipelined reduction operation. Now, the heads of the in-groups are ready to send the tentative distances of nodes in $B[i]$ to the heads of the out-groups. The analysis of these tree-operations is analogous the analysis for the out-groups.

6 Conclusion

The parameters governing the performance of Δ-stepping are the maximum path weight d_c and the largest step width Δ which ensures that there is only a linear number of Δ-connections (plus a light edge), n_Δ ($n_{\Delta+}$). If we want to introduce shortcuts efficiently, the choice of Δ must also bound the number of simple Δ-paths (plus a light edge), n'_Δ ($n'_{\Delta+}$). For parallelization, the corresponding l'_Δ has some influence too: On a CRCW PRAM our new algorithm with shortcut insertion needs $\mathcal{O}((l'_\Delta + \frac{d_c}{\Delta})\log n)$ time and $\mathcal{O}(m + n'_{\Delta+})$ work whp.

We now instantiate the result for some input graph classes. As a role model we look at general graphs with maximum in-degree and out-degree d and *random edge weights*, uniformly distributed[7] in the interval $[0,1]$. For $\Delta = \Theta(1/d)$ we have $l'_\Delta = \mathcal{O}(\log n/\log\log n)$ whp and $\mathbf{E}[n_{\Delta+}] \le \mathbf{E}[|P_{2\Delta}|] = \mathcal{O}(n)$ [10]. Thus, we get expected parallel time $\mathcal{O}((dd_c + \log n)\log n)$ and linear work. For example, for r-dimensional meshes with random edge weights we have $d_c = \mathcal{O}(n^{1/r})$ and hence execution time $\mathcal{O}(n^{1/r}\log n)$ using linear work for any constant r.

For random graphs from $\mathcal{G}(n, d/n)$, i.e., with edge probability d/n and random edge weights the maximum path weight is $d_c = \mathcal{O}(\log n/d)$ whp [10]. Thus, with our new approach we get an $\mathcal{O}(\log^2 n)$ parallel time linear expected work PRAM algorithm. This is a factor $\Theta(\log n/\log\log n)$ better than the best previously known work efficient algorithm from our earlier paper [10].

Another example are random geometric graphs $G_n(r)$ where n nodes are randomly placed in a unit square and each edge weight equals the Euclidean distances between the two involved nodes. An edge (u,v) is included if the Euclidean distance between u and v does not exceed the parameter $r \in [0,1]$. Random geometric graphs have been intensively studied since they are considered to be a relevant abstraction for many real world situations [14, 4]. Taking $r = \Theta(\sqrt{\log(n)/n})$ results in a connected graph with $m = \Theta(n\log n)$ edges and

[7] The results carry over to some other random distributions, too.

$d_c = \mathcal{O}(1)$ whp. For $\Delta = r$ the graph already comprises all relevant Δ-shortcuts such that we do not have to explicitly insert them. Consequently our PRAM algorithm runs in $\mathcal{O}((1/r)\log n)$ parallel time and performs $\mathcal{O}(n+m)$ work whp.

Acknowledgements

We would like to thank in particular Hannah Bast, Kurt Mehlhorn and Volker Priebe for many fruitful discussions and suggestions. Hannah Bast also pointed out the elegant solution of using a perfect hash function for semi-sorting requests.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : theory, algorithms and applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.
2. H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *3rd Symposium on Parallel Algorithms and Architectures*, pages 50–61, 1991.
3. Edith Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, September 1996.
4. J. Diaz, J. Petit, and M. Serna. Random geometric problems on $[0, 1]^2$. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 294–306. Springer, 1998.
5. E.W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.
6. E. A. Dinic. Economical algorithms for finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44, 1978.
7. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31, 1988.
8. Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 353–362, San Diego, CA, USA, June 1992. ACM Press.
9. Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, 1992.
10. U. Meyer and P. Sanders. Δ-stepping: A parallel shortest path algorithm. In *6th European Symposium on Algorithms (ESA)*, number 1461 in LNCS, pages 393–404. Springer, 1998.
11. U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. `http://www.mpi-sb.mpg.de/~sanders/papers/long-delta.ps.gz`, 1999.
12. J. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.
13. S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.
14. R. Sedgewick and J. S. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1:31–48, 1986.
15. Jesper Larsson Träff and Christos D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel algorithms for irregularly structured problems : Intern. workshop (IRREGULAR-3)*, volume LNCS 1117, pages 183–194S., Berlin, 1996. Springer.