Δ-Stepping : A Parallel Single Source Shortest Path Algorithm

Ulrich Meyer, Peter Sanders

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

Abstract. In spite of intensive research, little progress has been made towards fast and work-efficient parallel algorithms for the single source shortest path problem. Our *Δ-stepping algorithm*, a generalization of Dial's algorithm and the Bellman-Ford algorithm, improves this situation at least in the following "average-case" sense: For random directed graphs with edge probability $\frac{d}{n}$ and uniformly distributed edge weights a PRAM version works in expected time $\mathcal{O}(\log^3 n / \log\log n)$ using linear work. The algorithm also allows for efficient adaptation to distributed memory machines. Implementations show that our approach works on real machines. As a side effect, we get a simple linear time sequential algorithm for a large class of not necessarily random directed graphs with random edge weights.

1 Introduction

The shortest path problem is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [2]. Let $G = (V, E)$ be a directed graph, $|E| = m$, $|V| = n$, 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 *single source shortest path problem* (SSSP) is that of computing, for each vertex v reachable from s, the weight of a minimum-weight path from s to 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 [11]. Using Fibonacci heaps its running time is bounded by $\mathcal{O}(n \log n + m)$ [1]. Dijkstra's algorithm is inherently sequential since its work efficiency depends on the fact that nodes are considered in a fixed priority order. On the other hand, the Bellmann-Ford algorithm allows to consider all nodes in parallel but for that very reason it is not work efficient.

Therefore, we propose the following generalization named Δ-stepping: Nodes are ordered using *buckets* representing priority ranges of size Δ. Each bucket may

[1] There is also an $\mathcal{O}(n + m)$ time RAM algorithm for undirected graphs [25] which requires $n > 2^{12^{20}}$ due to the usage of atomic heaps however.

be processed in parallel. This can be implemented using only linear work if Δ is not too small. Refer to Sect. 2 for details.

From now on, with *random edge weights* we mean uniformly distributed weights in $[0,1]$ and random graphs are chosen from the set $\mathcal{G}(n, \frac{d}{n})$, i.e, have n nodes and edge probability $\frac{d}{n}$. In Sect. 3 we show that choosing $\Delta = \Theta\left(\frac{1}{d}\right)$ will not significantly increase the work performed and that only $\mathcal{O}\left(\log^2 n/\log\log n\right)$ buckets have to be emptied to solve the SSSP on random graphs. Then, in sections 4 and 5, Δ-stepping is adapted to the arbitrary CRCW-PRAM[2] model and to distributed memory machines respectively such that the work remains linear and time $\mathcal{O}\left(\log^3 n/\log\log n\right)$ suffices for random graphs. Sect. 6 complements the theoretical analysis with simulation and implementation results. Finally, Sect. 7 summarizes the most important aspects and sketches some possible future improvements.

Previous Work

The SSSP problem has so far resisted fast efficient parallel solutions: Most previous work was done in the PRAM model. The *work* of a parallel algorithm is given by the product of its running time and the number of processing units (PUs) p. There is no parallel $\mathcal{O}(n\log n + m)$ work PRAM algorithm with sublinear running time for general digraphs with non-negative edge weights. The best $\mathcal{O}(n\log n + m)$ work solution (refining [22] with [13]) has running time $\mathcal{O}(n\log n)$. All known algorithms with polylogarithmic execution time are work-inefficient. (The algorithm in [16] uses $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3(\log\log n/\log n)^{1/3})$ work.) An $\mathcal{O}(n)$ time algorithm requiring $\mathcal{O}((n+m)\log n)$ work was presented in [4].

For special classes of graphs, like planar digraphs [27] or graphs with separator decomposition [8], some less inefficient algorithms are known. Randomization was used in order to find approximate solutions with small error probability [19, 7]. For random graphs only unit weight edges have been considered so far [6]: This solution is restricted to constant edge probabilities or edge probability $\Theta(\log^k n/n)$ $(k > 1)$. In the latter case $\mathcal{O}(n\log^{k+1} n)$ work is needed.

Experimental studies for distributed computation of SSSP with multiple queues and some forms of oblivious parallel expansion strategies are provided in [5, 1, 26]. No profound theoretical analysis beyond correctness proofs are given, the experimental results yield only quite limited speedup.

2 The Δ-Stepping Algorithm

Our algorithm can be viewed as a variant of Dijkstra's algorithm. 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 $\mathrm{dist}(v)$, the weight of the shortest path from s to v. For unreached nodes, $\mathrm{tent}(v) = \infty$. Initially, s is

[2] Concurrent read concurrent write parallel random access machine [18].

queued, $\text{tent}(s) = 0$, 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., $\text{tent}(w)$ is set to $\min\{\text{tent}(w), \text{tent}(v) + c(v, w)\}$. 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.

```
foreach v ∈ V do                                          -- Initialize node data structures
    heavy(v) := {(v, w) ∈ E : c(v, w) > Δ}                                -- Find heavy edges
    light (v) := {(v, w) ∈ E : c(v, w) ≤ Δ}                               -- Find light edges
    tent  (v) := ∞                                                              -- Unreached
relax(s, 0); i := 0                                                 -- Source node at distance 0
while ¬isEmpty(B) do                                                   -- Some queued nodes left
    S := ∅                                                -- No nodes deleted for this bucket yet
    while B[i] ≠ ∅ do                                                             -- New phase
        Req := {(w, tent(v) + c(v, w)) : v ∈ B[i] ∧ (v, w) ∈ light(v)}
        S := S ∪ B[i]; B[i] := ∅                                       -- Remember deleted nodes
        foreach (v, x) ∈ Req do relax(v, x)                            -- This may reinsert nodes
    od
    Req      := {(w, tent(v) + c(v, w)) : v ∈ S    ∧ (v, w) ∈ heavy(v)}
    foreach (v, x) ∈ Req do relax(v, x)                       -- Relax previously deferred edges
    i := i + 1                                                                  -- Next bucket

Procedure relax(v, x)                                                        -- Shorter path to v?
    if x < tent(v) then                                 -- Yes: decrease-key respectively insert
        B[⌊tent(v)/Δ⌋] := B[⌊tent(v)/Δ⌋] \ {v}                                -- Remove if present
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {v}                           -- Insert into new bucket
        tent(v) := x
```

Fig. 1. High level Δ-stepping SSSP algorithm.

In the Δ-stepping algorithm shown in Figure 1 we weaken the total ordering of the queue and only maintain an array B of *buckets* such that $B[i]$ stores $\{v \in V : v \text{ is queued and } \text{tent}(v) \in [i\Delta, (i+1)\Delta)\}$. In each *phase*, i.e., each iteration of the inner while-loop, we remove all nodes from the first nonempty bucket and relax all *light* edges ($c(e) \leq \Delta$) which might lead to new nodes for the current bucket. For the remaining *heavy* edges it is sufficient to relax them once and for all when a bucket finally remains empty. Deletion and edge relaxation for an entire bucket can be done in parallel and in arbitrary order as long as an individual relaxation is atomic. If we do not want to sequentially execute the requests for a given node we can additionally exploit that requests which won't change the tent() array can be ignored.

For integer weights and $\Delta = 1$, Δ-stepping coincides with Dial's implementation of Dijkstra's algorithm (e.g. [2, Sect. 4.6]). We can reuse the data structure used there. Buckets are implemented as doubly linked lists. Inserting or deleting a node, finding a bucket for a given tentative distance and skipping an empty bucket can be done in constant time. By cyclically reusing empty buckets we only need space for $\max_{e \in E} c(e)/\Delta$ buckets.

Our algorithm may remove nodes from the queue for which $\mathrm{dist}(v) < \mathrm{tent}(v)$ and hence may have to reinsert those nodes until they are finally settled ($\mathrm{dist}(v) = \mathrm{tent}(v)$). In fact, for $\Delta = \infty$ we get the Bellman-Ford algorithm. It has high parallelism since all edges can be relaxed in parallel but it may be quite inefficient compared to Dijkstra's algorithm. The idea behind Δ-stepping is to find an easily computable Δ which yields a good compromise between these two extremes. There are graphs for which there is no good compromise. But at least for random edge weights and in particular for random graphs with random edge weights we identify such a compromise in Sect. 3. In Sect. 7 we give additional evidence that similar algorithms can also yield useful parallelism for real world problems.

3 Analysis

Our analysis of the Δ-stepping algorithms proceeds in three stages in order to make the results adaptable to different graph classes. In Sect. 3.1 we analyze the number of phases needed and the number of reinsertions in terms of logical properties like the maximum path weight $d_c := \max\{\mathrm{dist}(v) : \mathrm{dist}(v) < \infty\}$ which make no assumptions on the class of graphs investigated. Sect. 3.2 analyzes these conditions for the case of random edge weights. Finally, Sect. 3.3 completes the analysis by additionally assuming random graphs from $\mathcal{G}(n, d/n)$.

3.1 Reinsertions and Progress

Let P_Δ denote the set of paths with weight at most Δ. P_Δ plays a key role in analyzing both the overhead and the progress of Δ-stepping. The overhead compared to Dijkstra's algorithm is due to *reinsertions* and *rerelaxations*, i.e., insertions of nodes which have previously been deleted and relaxation of their outgoing edges.

Lemma 1. *The total number of reinsertions is bounded by $|P_\Delta|$ and the total number of rerelaxations is bounded by $|P_{2\Delta}|$.*

Proof. We give an injective mapping from the set of reinsertions into P_Δ. Consider a node v which is reinserted in phase t. There must be a most recent phase $t' \leq t$ when v was deleted. Consider a shortest path $(s, \ldots, v', \ldots v)$ where v' is the first unsettled node on this path immediately before phase t'. In contrast to v, v' is settled by phase t', i.e., $v' \neq v$. Furthermore, $(v', \ldots, v) \in P_\Delta$ since both v and v' are deleted by phase t'. Since v' is settled, the reinsertion of v in phase t can be uniquely mapped to $(v', \ldots, v)$.

Similarly, each rerelaxation can be uniquely identified by a reinsertion plus a light edge e, i.e., the path in $P_{2\Delta}$ defined by appending e to the path in P_Δ identifying the reinsertion can be injectively mapped to the rerelaxation. ■

The number of phases needed can be bounded based on the step width Δ, the maximum distance d_c and a parameter $l_{\max}$ which must exceed the maximum number of edges in any path in P_Δ.

Lemma 2. *For any step width Δ, the number of phases is bounded by $\frac{d_c}{\Delta} l_{\max}$.*

Proof. It suffices to show that no bucket is expanded more than $l_{\max}$ times. So assume the contrary, i.e., there is a bucket $B[i]$ which is expanded at time steps $t - l_{\max}, \ldots, t$. Let $v = \min\{\text{tent}(w) : w \in B[i](t)\}$. v is removed for the last time in phase t because otherwise it could not be the minimum queued element. Consider the path $P = (s, \ldots, v_{l_{\max}}, \ldots, v_1, v)$ in the shortest path tree leading from s to v. v_1 must be settled in phase $t-1$. It cannot be settled later because then v would not have its final distance yet. v_1 cannot be settled earlier because else v would have received its final distance earlier and would also be settled by now since it is already in $B[i]$ for the last $l_{\max}$ phases.

Similarly, we can show by induction that $v_1, \ldots, v_{l_{\max}}$, have been settled one by one in the last $l_{\max}$ phases. So, $v_{l_{\max}}$ has been settled in phase $t - l_{\max}$ and this is only possible if its final distance puts it in bucket $B[i]$ and therefore $(v_{l_{\max}}, \ldots, v)$ is in P_Δ and has $l_{\max}$ edges. This is a contradiction. ∎

For a parallel algorithm the goal is to find a Δ which is small enough to keep $|P_\Delta|$ in $\mathcal{O}(n)$ yet large enough to yield sufficient parallelism. For example, it is easy to see that there are no reinsertions if we choose the step-width Δ to be the minimum edge weight and $r = \frac{d_c}{\Delta}$ phases suffice in this case [12]. We can even achieve the same result for larger Δ by adding a "shortcut" edge (v, w) for each $(v, \ldots, w) \in P_\Delta$. However, our experimental results on random graphs indicate that the basic algorithm also completes in $\mathcal{O}(\frac{d_c}{\Delta})$ phases in practice so that we continue with the analysis of the simpler algorithm.

3.2 Random Edge Weights

This section is devoted to proving that $\Delta = \Theta(1/d)$ is a good compromise between work efficiency and parallelism for random edge weights:

Theorem 1. *Given a graph with maximum degree d or a random graph from $\mathcal{G}(n, d/n)$. For random edge weights, a $\Theta(1/d)$-stepping scheme performs $\mathcal{O}(dn)$ expected sequential work divided between $\mathcal{O}(\frac{d_c}{\Delta} \cdot \frac{\log n}{\log \log n})$ phases whhp*[3].

Our main tool is the fact that long paths with small weight are unlikely.

Lemma 3. *Given a path of length l. The probability that its total weight is bounded by $\Delta \leq 1$ is $\Delta^l / l!$.*

Proof. By induction over l. ∎

This is the only part in our analysis which would have to be adapted for other than uniform weight distribution. Now we can bound $|P_\Delta|$ and the $l_{\max}$ from Lemma 2.

Lemma 4. *For $\Delta = \Theta(1/d)$, $\mathbf{E}[|P_\Delta|] = \mathcal{O}(n)$, $\mathbf{E}[|P_{2\Delta}|] = \mathcal{O}(n)$ and $l_{\max} = \mathcal{O}(\log n / \log \log n)$ whp.*

[3] Throughout this paper "whp" stands for "with high probability" in the sense that the probability for some event is at least $1 - n^{-\beta}$ for a constant $\beta > 0$.

Proof. There can be at most d^l paths of length l leading into a given node v or nd^l such paths overall.(For random graphs there are $\leq n^l$ possible paths per node with a probability of $(d/n)^l$ each.) Using Lemma 3 we can conclude that the expected number of light paths is bounded by $\sum_{l\geq 1} nd^l \Delta^l/l! = n(e^{d\Delta}-1) = \mathcal{O}(n)$ and analogously for $P_{2\Delta}$.

Similarly, $\mathbf{P}\left[\exists P \in P_\Delta : |P| = l\right] \leq n(d\Delta)^l/l!$. Therefore

$$\begin{aligned}\mathbf{P}\left[\exists P \in P_\Delta : |P| \geq l_{\max}\right] &\leq \sum_{l\geq l_{\max}} n(d\Delta)^l/l! \leq n(d\Delta)^{l_{\max}}/l_{\max}! \sum_{l\geq 0}(d\Delta)^l/l! \\ &= n(d\Delta)^{l_{\max}} e^{d\Delta}/l_{\max}! = \mathcal{O}(n)\,/l_{\max}! \text{ for } \Delta \leq 1/d \\ &\leq (e/l_{\max})^{l_{\max}} \cdot \mathcal{O}(n) \text{ since } k! \geq (k/e)^k \ .\end{aligned}$$

Now it is easy to see that we can choose an $l_{\max} = \mathcal{O}(\log n/\log\log n)$ such that the above probability is polynomially small. ■

Theorem 1 is now an immediate consequence of lemmata 1, 2 and 4.

3.3 Random Graphs

So far, the analysis treated the maximum path weight, d_c, as a parameter and it is clear that there are graphs – even with random edge weights – where $d_c = \Omega(n)$ so that it makes no sense to expand nodes in parallel. But this is quite untypical. In particular, for random graphs d_c is rather small:

Theorem 2. *For random graphs from $\mathcal{G}(n, d/n)$, $d_c = \mathcal{O}\left(\frac{\log n}{d}\right)$ whp.*

For large degree – $d \geq a \log n$ for some constant a – this is a well known result [17, 14]. We now outline a proof for the remaining case $d = \mathcal{O}(\log n)$ and refer to [10] for more details.

From random graph theory we know that if s is not in the giant component it is in a very small component of size $\mathcal{O}(\log n)$ and the SSSP is very simple. Otherwise d_c can be bounded by the diameter of the giant component.[4]

Lemma 5. *For $d > 1$ the diameter of the giant component is $\mathcal{O}(\log n)$ whp.*

Proof. (Outline) The node exploration procedure used in [3, Sect. 10] can be adapted to a breadth first traversal. Using this approach it can be shown that all but $\mathcal{O}(n^{1/2+\epsilon})$ nodes in the giant component are reached after $\mathcal{O}(\log n)$ phases whp. Later, the expected number of newly reached nodes in the breadth first traversal decreases geometrically so that after $\mathcal{O}(\log n)$ more steps no more nodes are reached whp. ■

[4] Note that this diameter is *not* the diameter of the graph (which may be infinite in this "sparse" case). So, the fact that the diameter of random graphs is well studied in the literature is not directly helpful.

This yields a good upper bound for constant d. For larger degree, the basic idea is to first consider only those edges with weight at most $2/d$ (any a/n with $a > 1$ will do). Those form a "backbone" of the giant component for which $d_c = \mathcal{O}(\frac{\log n}{d})$. Finally, the exploration procedure from [3, Sect. 10] can be adapted once more to show that every node in the giant component connects to the backbone by a path of length $\mathcal{O}(\lceil \frac{\log n}{d} \rceil)$ whp. ■

Substituting this result into Theorem 1 we see that $r = \mathcal{O}\left(\log^2 n / \log\log n\right)$ phases of a $\Theta(1/d)$-stepping algorithm suffice to solve the SSSP. If we have introduced shortcut edges this reduces to $\Theta(\log n)$ phases.

4 Adaptation to PRAM

We now explain how the abstract $\Theta(1/d)$-stepping algorithm from Fig. 1 can be efficiently implemented on an arbitrary-write CRCW PRAM for random graphs from $\mathcal{G}(n, \frac{d}{n})$ with random edge weights. The actual number of edges is $m = \Theta(dn)$ whp. We concentrate on the most interesting case $d = \mathcal{O}(\log n)$. (Note that whp all but the $c \log n$ smallest edges per node can be ignored without changing the shortest paths for some constant c [17, 14].) A direct way for handling larger m is discussed in [10].

Preparations: The nodes are assigned to random PUs by generating an array "ind" of random PU indices. The adjacency lists are reorganized into arrays of heavy and light edges for each node. Each of the p PUs maintains its own bucket structure and stores there the queued nodes it is responsible for. These operations can be done in time $\mathcal{O}(dn/p + \log(dn))$ and take $\mathcal{O}(dn)$ space.

Loop Control: Detecting when one or all buckets are globally empty and advancing i is easy to do in time $\mathcal{O}(\log n)$ (or even $\mathcal{O}(1)$) per iteration.

Maintaining S: S can be represented as a simple list per PU. Inserting a node several times can be avoided by storing a flag with each node which is set when it is inserted for the first time.

Generating Requests: Since the nodes have been randomly assigned, no PU has to delete more than $\mathcal{O}(|B[i]|/p + \log n)$ nodes from its local part of $B[i]$ whp. Using prefix sums, the light edges to be scanned can be evenly distributed between the PUs in time $\mathcal{O}(\log n)$. The request set generated is represented as a global array of target-distance pairs. Analogously, requests for heavy edges can be generated in a load balanced way with a control overhead of $\mathcal{O}(\log n)$ time steps whp.

Assigning Heavy Requests to PUs: PU i maintains a request buffer which must be a constant factor larger than needed to accommodate the requests (w, x) with $\mathrm{ind}(w) = i$. Since heavy edges are relaxed only once, for random graphs their targets are independently distributed and therefore by Chernoff bounds, a buffer area of size $\mathcal{O}(|S|/p + \log n)$ sufficies whp. The requests can be placed into the appropriate buffers using randomized dart throwing in time $\mathcal{O}(|B[i]|/p + \log n)$ whp [21]. (For the unlikely case that a buffer is to small correctness can be preserved by checking periodically whether the dart throwing has terminated and increasing the buffer sizes if necessary.)

Assigning Light Requests to PUs: Assigning light request works as for heavy requests. The targets of rerelaxed edges are no longer independent. However, targets are still independent when edges are relaxed for the first time. Let $K_i := |\{(v,w) \in E : \mathrm{dist}(v) \in [i\Delta, (i+1)\Delta) \wedge c((v,w)) \leq \Delta\}|$, i.e., the number of light edges ever relaxed in bucket i not counting rerelaxations. Then, by Chernoff bounds, no node receives more than $\mathcal{O}(\lceil K_i \log n/n \rceil)$ requests in any phase for bucket i. Let K'_{ij} denote the number of requests sent in the j-th phase for bucket i. Since nodes are placed independent of the computation, we can use the weighted Chernoff bound from [23], to see that no PU receives more than $\mathcal{O}\left(K'_{ij}/p + \log n \lceil K_i \log n/n \rceil\right)$ requests in phase j for bucket i whp. By Lemma 4 we have $\mathbf{E}[\sum_{ij} K'_{ij}] = \mathcal{O}(n)$. Furthermore, no bucket is emptied more than $l_{\max} = \mathcal{O}(\log n / \log\log n)$ times so that $\sum_{ij} K_i = \mathcal{O}(n l_{\max})$ whp. Therefore, the expected request contention summed over all phases is bounded by $\mathcal{O}\left(n/p + l_{\max} \log^2 n\right)$.

Performing Relaxations: Each PUs scans its request buffer and sequentially performs the relaxations assigned to it. Since no other PUs work on its nodes the relaxations will be atomic. (This is the only place where significant modifications for handling the case $d \gg \log n$ would be needed if one chooses not to filter out too heavy edges in a preprocessing: p/n PUs work together on a single node and first find the minimum distance request and only perform a relaxation for this request.)

To summarize, control overhead accounts for at most $\mathcal{O}(\log n)$ time per phase; the expected load imbalance accounts for at most $\mathcal{O}(\log n)$ time per phase plus $\mathcal{O}\left(n/p + l_{\max} \log^2 n\right)$ overall for light requests; work done in a load balanced way cannot take more than $\mathcal{O}(dn/p)$ expected time.[5]

Theorem 3. *The SSSP on random graphs from $\mathcal{G}(n, \frac{d}{n})$ with random edge weights can be solved in expected time $\mathcal{O}(\log^3 n / \log\log n)$ and $\mathcal{O}(dn)$ work using $\frac{dn \log\log n}{\log^3 n}$ processors on a CRCW PRAM.*

5 Adaptation to Distributed Memory

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. The analysis can focus on finding the number of necessary basic operations. The execution time for a particular network or abstract model is then easy to determine. For example, in the BSP model [20] we simply substitute $T_{\mathtt{routing}}(k) = \mathcal{O}(l + (k + \log p)g)$ and $T_{\mathtt{coll}}(k) = \mathcal{O}((l + gk)\log p)$.

Let $r = \mathcal{O}\left(\log^2 n / \log\log n\right)$ denote the number of phases required whp. Our PRAM algorithm is already almost a distributed memory algorithm if we restrict ourselves to the practically most interesting case $p \leq \frac{n}{r \log n}$. Each PU stores the

[5] All results also hold with high probability if such a bound for the number of rerelaxations is available.

adjacency lists of the nodes assigned to it using a hash function $\mathrm{ind}(w)$ we assume to be computable in constant time. (Essentially the same assumptions are made for efficient PRAM simulation algorithms [28, Section 4.3] and this is certainly warranted for the simple hash functions used in practice.) Load balancing for generating requests is already achieved if each PU simply scans the adjacency lists of its local part of $B[i]$ (for random graphs).

The dart throwing process for assigning requests can be replaced by simply routing a request (w, x) to PU $\mathrm{ind}(w)$. An analysis similar to the PRAM case yields an expected execution time in $\mathcal{O}\big(dn/p + r(T_{\mathtt{coll}}(1) + T_{\mathtt{routing}}(dn/(pr)))\big)$.

Many edges Using some additional measures we can employ more PUs ($p = \frac{m}{r \log n}$ as in the PRAM case). Again we concentrate on the case $d = \mathcal{O}(\log n)$. Now d PUs work together in p/d *list groups*. As before, nodes are hashed to individual PUs. But the heavy parts of the adjacency lists are independently assigned to a random list group where they are stored in a global round robin fashion: looking at a particular list of length l, each PU in the corresponding list group will either store $\lfloor l/d \rfloor$ or $\lceil l/d \rceil$ entries of that list. When heavy edges are relaxed, a PU sends the distances of the nodes it has deleted to the first PU in the list group responsible for this node. The first PU in a list group transmits all the distance node pairs it has received to the other group members using a pipelined broadcast operation. After a detailed analysis and similar measures as in the PRAM case for $d \gg \log n$ we get:

Theorem 4. *If the number of delete-phases is bounded by r then the SSSP on random graphs with random edge weights can be solved on a distributed memory machine with $p \leq \frac{dn}{r \log n}$ processors in expected time $\mathcal{O}(\frac{dn}{p} + r(T_{\mathtt{coll}}(\frac{dn}{pr}) + T_{\mathtt{routing}}(\frac{dn}{pr})))$ and $\mathcal{O}(dn)$ work.*

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 that we get the same asymptotic performance as our CRCW-PRAM algorithm.

6 Simulation and Implementation

Simulations of different algorithm variants played a key role in designing the algorithm and are still interesting as a means to estimate the constant factors involved. $\Delta = 4/d$ proves to be a good choice for the bucket range: For all tested values for $d \geq 2$ the number of phases were bounded by $5 \ln n$ and less than $0.25n$ reinsertions occured. Reachable nodes could be accessed by paths of weight $2.15 \frac{\ln n}{d}$ or shorter. For sparse graphs, the number of phases can be further reduced at a low price in terms of reinsertions by emptying all buckets at once when they hold only few nodes.

Actually implementing a linear work algorithm which requires a linear number of tiny messages with irregular communication pattern is not easy. However, for small p and large n, a machine with high bandwidth interconnection network and an efficient library routine for personalized all-to-all communication can do the job. We implemented a simple version of the algorithm for distributed memory machines and random d-regular graphs using the library MPI [24]. Tests were run on an INTEL Paragon with 16 processors. For $n = 2^{19}$ nodes and $d = 3$ speedup 9.2 was obtained against the sequential Δ-stepping approach. The latter in turn is 3.1 times faster than an optimized sequential implementation of Dijkstra's algorithm. Due to the increased communication costs, our results on dense graphs are slightly worse: for $n = 2^{16}$ and $d = 32$ the speedup of parallel Δ-stepping compared to its sequential counterpart was 7.5 [6], sequential Δ-stepping was 1.8 times faster than Dijkstra's algorithm.

7 Conclusions and Future Work

A $\Theta(1/d)$-stepping scheme solves the SSSP for random graphs from $\mathcal{G}(n, d/n)$ with random edge weights in $\mathcal{O}(\log^3 n/\log\log n)$ time and $\mathcal{O}(dn)$ expected work on PRAMs and many distributed memory machines. If one views random graphs with random edge weights as a model for "average" graphs this implies a striking difference between the average case and the worst case for which no sublinear time work efficient solution is known.

The number of phases can be reduced by a factor $\Theta(\log n/\log\log n)$ if shortcut edges for paths in $P_{\Theta(1/d)}$ are inserted. Following our simulations we conjecture that even without shortcuts $\mathcal{O}(\log n)$ phases suffice. It might be possible to further speed up the CRCW-PRAM algorithm by replacing our $\mathcal{O}(\log n)$ load balancing and dart throwing routines by "almost constant time algorithms" [15]. For $d = \Omega(\log^* n)$ this might even be possible in a work efficient way.

Our SSSP solution immediately yields an improved algorithm for the all-pairs shortest path problem on random graphs with random edge weights.

For random graphs with random edge weights the random assignment of nodes to PEs should be dispensable. Since this is the only source of randomness needed in the distributed memory algorithm, we get a deterministic algorithm. From a practical point of view it is more interesting however to lift the randomness assumptions on the graph and the edge weights.

For arbitrary directed graphs with random edge weights and maximum degree d sequential $\Theta(1/d)$-stepping works in expected time $\mathcal{O}(d_c d + m + n)$ where d_c is the maximum shortest path weight. This is linear for all but quite degenerate cases. If $d_c d \ll m$ there is also considerable parallelism which can be exploited without affecting work efficiency. However, there is some work to be done regarding an efficient parallel implementation for graphs where load balancing is more difficult than for random graphs without shortcuts.

Interesting research can be done on parallel shortest path for non-random edge weights. One approach could be to start with $\Delta = \min_{e\in E} c(e)$ and then double Δ until $|P_\Delta| \geq n$. $|P_\Delta|$ can for example be determined using parallel

[6] Our current implementation does not distinguish between heavy and light edges which increases the communication overhead. Therefore, we expect somewhat higher speedups for the full version of the paper.

depth first traversal of light paths starting from each node. However, this is not work efficient for small d.

One can also look for other ways of determining the set R of nodes to be deleted in a phase. We have made experiments where $|R|$ is some fraction of the total priority queue size $|Q|$. In our simulations this works as well as $\Theta(1/d)$-stepping for random graphs with $|R| = \Theta(|Q| / \log |Q|)$, for random planar graphs we could even use $|R| = |Q| / 2$. We also tested this approach on real world graphs and edge weights: starting with a road-map of a town ($n = 10,000$) the tested graphs successively grew up to a large road-map of Southern Germany ($n = 157,457$). Good performance was found for $|R| = \Theta(|Q|^{3/4})$. While repeatedly doubling the number of nodes, the average number of phases (for different starting points) only increased by a factor of about 1.5; for $n = 157,457$ the simulation needed $1,178$ phases, the number of reinserts was bounded by $0.2n$. In [10, 9] we develop an algorithm which needs no reexpansions for arbitrary edge weights. However, even for random edge weights it needs $\Theta\left(n^{1/3}\right)$ phases even for random edge weights.

Acknowledgements

We would like to thank in particular Kurt Mehlhorn and Volker Priebe for many fruitful discussions and suggestions.

References

1. P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest path problem. *International Journal of Parallel Programming*, 20(4):271–298, 1991.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms and Applications*. Prentice Hall, 1993.
3. N. Alon, J. H. Spencer, and P. Erdős. *The Probabilistic Method*. Wiley, 1992.
4. G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operation. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 689–693. IEEE, 1997.
5. K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.
6. A. Clementi, J. Rolim, and E. Urland. Randomized parallel algorithms. In A. Ferreira and P. Pardalos, editors, *Solving Combinatorial Optimization Problem in Parallel*, volume 1054 of *LNCS*, pages 25–50. Springer, 1996.
7. E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 16–26, 1994.
8. E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.
9. 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.

10. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. Parallelizing Dijkstra's shortest path algorithm. Technical report, MPI-Informatik, 1998. in preparation.
11. E. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.
12. E. A. Dinic. Economical algorithms for finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44, 1978.
13. 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(11):1343–1354, 1988.
14. A. Frieze and G. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Appl. Math.*, 10:57–77, 1985.
15. T. Hagerup. The log-star revolution. In A. Finkel and M. Jantzen, editors, *Proceedings of Symposion on Theoretical Aspects of Computer Science (STACS '92)*, volume 577 of *LNCS*, pages 259–280. Springer, Feb. 1992.
16. Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pairs shortest paths in directed graphs. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 353–362. ACM Press, 1992.
17. R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Math. Oper. Res.*, 10(4):557–564, 1985.
18. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
19. P. N. Klein and S. Sairam. A parallel randomized approximation scheme for shortest paths. In *Proc. 24th Ann. ACM Symp. on Theory of Computing*, pages 750–758, Victoria, B.C., Canada, 1992.
20. W. F. McColl. Universal computing. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. Euro-Par '96 Parallel Processing*, volume 1123 of *LNCS*, pages 25–36. Springer, 1996.
21. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489. IEEE, 1985.
22. R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing*, pages 14–20. IEEE, 1985.
23. P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.
24. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference*. MIT Press, 1996.
25. M. Thorup. Undirected single source shortest paths in linear time. In *38th Annual Symposium on Foundations of Computer Science*, pages 12–21. IEEE, 1997.
26. J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.
27. J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Irregular' 96*, volume 1117 of *LNCS*, pages 183–194. Springer, 1996.
28. L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 943–971. Elsevier, 1990.