A Parallelization of Dijkstra's Shortest Path Algorithm

A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders

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

Abstract. The single source shortest path (SSSP) problem lacks parallel solutions which are fast and simultaneously work-efficient. We propose simple criteria which divide Dijkstra's sequential SSSP algorithm into a number of phases, such that the operations within a phase can be done in parallel. We give a PRAM algorithm based on these criteria and analyze its performance on random digraphs with random edge weights uniformly distributed in $[0, 1]$. We use the $\mathcal{G}(n, d/n)$ model: the graph consists of n nodes and each edge is chosen with probability d/n. Our PRAM algorithm needs $\mathcal{O}(n^{1/3} \log n)$ time and $\mathcal{O}(n \log n + dn)$ work with high probability (whp). We also give extensions to external memory computation. Simulations show the applicability of our approach even on non-random graphs.

1 Introduction

Computing shortest paths is an important combinatorial optimization problem with numerous applications. 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 $\text{dist}(v)$ 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 digraphs with non-negative edge weights is Dijkstra's algorithm [8]. Using Fibonacci heaps its running time is $\mathcal{O}(n \log n + m)$[1]. Dijkstra's algorithm maintains a partition of V into *settled*, *queued* and *unreached* nodes and for each node v a *tentative distance* $\text{tent}(v)$; $\text{tent}(v)$ is always the weight of some path from s to v and hence an upper bound on $\text{dist}(v)$. For unreached nodes, $\text{tent}(v) = \infty$. Initially, s is 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 $\min\{\text{tent}(w), \text{tent}(v) + c(v, w)\}$.

[1] There is also an $\mathcal{O}(n + m)$ time algorithm for undirected graphs [20], but it requires the RAM model instead of the comparison model which is used in this work.

Luboš Prim et al. (Eds.): MFCS'98, LNCS 1450, pp. 722–731, 1998.

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 queue may contain more than one node v with $\text{tent}(v) = \text{dist}(v)$. All such nodes could be removed simultaneously, the problem is to identify them. In Sect. 2 we give simple sufficient criteria for a queued node v to satisfy $\text{tent}(v) = \text{dist}(v)$. We remove all nodes satisfying the criteria simultaneously.

Although there exist worst-case inputs needing $\Theta(n)$ phases, our approach yields considerable parallelism on random directed graphs: We use the random graph model $\mathcal{G}(n, d/n)$, i.e., there are n nodes and each theoretically possible edge is included into the graph with probability d/n. Furthermore, we assume random edge weights uniformly distributed in $[0, 1]$: In Sect. 3 we show that the number of phases is $\mathcal{O}(\sqrt{n})$ using a simple criterion, and $\mathcal{O}(n^{1/3})$ for a more refined criterion with high probability (whp)[2].

Sect. 4 presents an adaption of the phase driven approach to the CRCW PRAM model which allows p processors (PUs) concurrent read/write access to a shared memory in unit cost (e.g. [13]). We propose an algorithm for random graphs with random edge weights that runs in $\mathcal{O}(n^{1/3} \log n)$ time whp. The work, i.e., the product of its running time and the number of processors, is bounded by $\mathcal{O}(n \log n + dn)$ whp.

In Sect. 5 we adapt the basic idea to external memory computation (I/O model [22]) where one assumes large data structures to reside on D disks. In each I/O operation, D *blocks* from distinct disks, each of size B, can be accessed in parallel. We derive an algorithm which needs $\mathcal{O}(\frac{n}{D} + \frac{dn}{DB} \log_{S/B} \frac{dn}{DB})$ I/Os on random graphs whp and can use up to $D = \mathcal{O}(\min\{n^{2/3}/\log n, \frac{S}{B}\})$ independent disks. S denotes the size of the internal memory.

In Sect. 6 we report on simulations concerning the number of phases needed for both random graphs and real world data. Finally, Sect. 7 summarizes the results and sketches some open problems and future improvements.

Previous Work

PRAM algorithms: 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 [9] has running time $\mathcal{O}(n \log n)$. All known algorithms with polylogarithmic execution time are work-inefficient. ($\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3 (\log \log n / \log n)^{1/3})$ work for the algorithm in [11].) An $\mathcal{O}(n)$ time algorithm requiring $\mathcal{O}((n + m) \log n)$ work was presented in [3].

For special classes of graphs, like planar digraphs [21] or graphs with separator decomposition [6], more efficient algorithms are known. Randomization was used in order to find approximate solutions [5]. Random graphs with *unit* weight edges are considered in [4]. The solution is restricted to dense graphs ($d = \Theta(n)$) or edge probability $d = \Theta(\log^k n/n)$ ($k > 1$). In the latter case $\mathcal{O}(n \log^{k+1} n)$ work is needed. Properties of shortest paths in complete graphs ($d = n$) with

[2] 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 $\beta > 0$.

random edge weights are investigated in [10, 12]. In contrast to all previous work on random graphs, we are most interested in the case of small, even constant d. **External Memory:** The best result on SSSP was published in [16]. This algorithm requires $\mathcal{O}(n + \frac{m}{DB} \log_2 \frac{m}{B})$ I/Os. The solution is only suitable for small n because it needs $\Theta(n)$ I/Os.

2 Running Dijkstra's Algorithm in Phases

We give several criteria for dividing the execution of Dijkstra's algorithm into phases. In the first variant (OUT-version) we compute a threshold defined via the weights of the *outgoing* edges: let $L = \min\{\text{tent}(u) + c(u,z) : u \text{ is queued and } (u,z) \in E\}$ and remove all nodes v from the queue which satisfy $\text{tent}(v) \leq L$. Note that when v is removed from the queue then $\text{dist}(v) = \text{tent}(v)$. The threshold for the OUT-criterion can either be computed via a second priority queue for $o(v) = \text{tent}(v) + \min\{c(v,u) : (v,u) \in E\}$ or even on the fly while removing nodes.

The second variant, the IN-version, is defined via the *incoming* edges: let $M = \min\{\text{tent}(u) : u \text{ is queued}\}$ and $i(v) = \text{tent}(v) - \min\{c(u,v) : (u,v) \in E\}$ for any queued vertex v. Then v can be safely removed from the queue if $i(v) \leq M$. Removable nodes of the IN-type can be found efficiently by using an additional priority queue for $i(\cdot)$.

Finally, the INOUT-version applies both criteria in conjunction.

3 The Number of Phases for Random Graphs

In this section we first investigate the number of delete-phases for the OUT-variant of Dijkstra's algorithm on random graphs. Then we sketch how to extend the analysis to the INOUT-approach. We start with mapping the OUT-approach to the analysis of the reachability problem as provided in [14] and [1, Sect. 10.5] and give lower bounds on the probability that many nodes can be removed from the queue during a phase.

Theorem 1. OUT-approach. *Given a random graph from $\mathcal{G}(n, d/n)$ with edge labels uniformly distributed in $[0,1]$, the SSSP problem can be solved using $r = \mathcal{O}(\sqrt{n})$ delete-phases with high probability.*

We review some facts of the reachability problem using the notation of [1].

The following procedure determines all nodes reachable from a given node s in a random graph G from $\mathcal{G}(n, d/n)$. Nodes will be neutral, active, or dead. Initially, s is active and all other nodes are neutral, let time $t = 0$, and $Y_0 = 1$ the number of active nodes. In every time unit we select an arbitrary active node v and check all theoretically possible edges (v, w), w neutral, for membership in G. If $(v, w) \in E$, w is made active, otherwise it stays neutral. After having treated all neutral w in that way, we declare v dead, and let Y_t equal the new number of active nodes. The process terminates when there are no active nodes.

The connection with the OUT-variant of Dijkstra's algorithm is easy: The distance labels determine the order in which queued vertices are considered and declared dead, and time is partitioned into intervals (=phases): If a phase of the OUT-variant removes k nodes this means that the time t increases by k.

Let Z_t be the number of nodes w that are reached for the first time at time t. Then $Y_0 = 1$, $Y_t = Y_{t-1} + Z_t - 1$ and $Z_t \sim B[n-(t-1)-Y_{t-1}, d/n]$ where $B[n,q]$ denotes the binomial distribution for n trials and success probability q.

Let T be the least t for which $Y_t = 0$. Then T is the number of nodes that are reachable from s. The recursive definition of Y_t is continued for all t, $0 \le t \le n$. We have $Y_t \sim B[n-1, 1-(1-d/n)^t] + 1 - t$.

It is shown in [1] that the number of nodes reachable from s is either very small (less than $\mathcal{O}(\log n)$) or concentrates around $T_0 = \alpha_0 n$, where $0 < \alpha_0 < 1$, and $\alpha_0 = 1 - e^{-d\alpha_0}$. Only the case $T \approx T_0$ requires analysis; if $T = \mathcal{O}(\log n)$ the number of phases is certainly small. Chernoff bounds yield:

Lemma 1. *Except for small t ($t \le \sqrt{n}$) and large t ($t \ge T_0 - n^{1/2+\epsilon}$) Y_t is $(1 \pm o(1/n^2))\mathbf{E}[Y_t]$ with high probability.*

The *yield* of a phase in the OUT-variant is the number of nodes that are removed in a phase. We call a phase starting at time t *profitable* if its yield is $\Omega(\sqrt{Y_t/d})$ and *highly profitable* if its yield is $\Omega(\sqrt{(Y_{t/2} - t/2)t/n})$ and show:

Lemma 2. *A phase is profitable with probability at least $1/8$. A phase starting at time t with $\frac{n \ln d}{d} \le t \le \alpha_0 n - n/d$ is highly profitable with probability at least $1/8$.*

Theorem 1 follows fairly easily from lemmas 1 and 2: We call a phase with starting time t *early extreme* if $t \le \sqrt{n}$, *early intermediate* if $\sqrt{n} < t \le (n \ln d)/d$, *early central* if $(n \ln d)/d < t \le n/2$, *late central* if $n/2 < t \le \alpha_0 n - n/d$, *late intermediate* if $\alpha_0 n - n/d < t \le \alpha_0 n - n^{1/2+\epsilon}$, and *late extreme* if $\alpha_0 n - n^{1/2+\epsilon} < t$, and show that there are only $\mathcal{O}(\sqrt{n})$ phases of each kind with high probability. Consider, for example, the late intermediate phases. A profitable late intermediate phase starting at time t has yield $\Omega(\sqrt{Y_t/d}) = \Omega(\sqrt{\mathbf{E}[Y_t]/d}) = \Omega(\sqrt{(\alpha_0 n - t)/d})$, where the first equality holds with high probability by Lemma 1. Let $t' := \alpha_0 n - t$. The number of profitable phases with $2^i \le t' < 2^{i+1}$ is therefore $\mathcal{O}(\sqrt{2^i d})$ and the number of profitable phases with $\alpha_0 n - n/d \le t = \alpha_0 n - t'$ is therefore $\sum_{i \le \log(n/d)} \mathcal{O}(\sqrt{2^i d}) = \mathcal{O}(\sqrt{n})$. Since a phase is profitable with probability at least $1/8$, the number of phases is also $\mathcal{O}(\sqrt{n})$ with high probability. The number of early extreme phases is $\mathcal{O}(\sqrt{n})$ trivially. For the number of late extreme phases we argue as follows. We first show that $T \le \alpha_0 n + n^{1/2+\epsilon}$ with high probability and then consider the first time t_1, $t_1 \ge \alpha_0 n - n^{1/2+\epsilon}$, with $Y_{t_1} \le n^{1/4}$. Lemma 1 implies that the number of late extreme phases starting before t_1 is $\mathcal{O}(\sqrt{n})$. If the number of phases starting after t_1 is $\sqrt{n}$ or more, then $Z_{t_1} + Z_{t_1+1} + \cdots + Z_{t_1+\sqrt{n}} \ge \sqrt{n} - n^{1/4} \ge \sqrt{n}/2$. The probability of this event is bounded by $\mathbf{P}\left[B[n^{1/2}(n-(n-n^{1/2+\epsilon})), d/n] \ge \sqrt{n}/2\right]$, which is exponentially small.

The idea for the proof of Lemma 2 is as follows. Let $v_1, v_2, \ldots, v_q$, $q = Y_t$, be the queued nodes in order of increasing tentative distances, and let L' be the value of L in the previous phase. The distance labels $\text{tent}(v_i)$ are random variables in $[L', L'+1]$. We show that their values are independent and their distributions are biased towards smaller values (since $\text{tent}(v_i) = \min\{\text{dist}(v) + c(v, v_i),\ v$ settled and $(v, v_i) \in E\}$, $\text{dist}(v) \le L'$, $c(v, v_i)$ uniform in $[0, 1]$. The value of $\text{tent}(v_r)$ is therefore less than r/q with constant probability for arbitrary r, $1 \le r \le q$. The number of edges out of $v_1, \ldots, v_r$ is $r(d/n)n = rd$ on the average and not much more with constant probability. The shortest of these edges has length about $\frac{1}{rd}$. We remove $v_1, \ldots, v_r$ from the queue if $\text{tent}(v_r)$ is smaller than the length of the shortest edge out of $v_1, \ldots, v_r$. This is the case (with constant probability) if $r/q \le \frac{1}{rd}$ or $r \le \sqrt{q/d}$.

For the phases starting at time t with $(n \ln d)/d \le t \le \alpha_0 n - n/d$ we refine the argument as follows. We call a node queued at time t *old* if it was already queued before time $t/2$ and show that the number of old queued nodes at time t is at least $Y_{t/2} - t/2$. Each old queued node has an expected indegree from settled nodes of at least $\frac{t}{2}\frac{d}{n}$. We use this fact to deduce that $\text{tent}(v_r)$ is less than $r/(\frac{td}{2n}(Y_{t/2} - t/2))$ with constant probability and then proceed as above.

INOUT Approach. If both IN- and OUT-criterion are applied together, the tentative distance labels of queued nodes may spread over a range as large as $[L', L'+2)$, while the edge weights are only in $[0, 1]$. In order to reuse the analysis of the OUT-part we analyze a slightly slower version which alternates the two criteria in the following way:

***I*-Step:** Let q be the current queue size. Apply the IN-criterion to the $g(q)$ nodes with smallest tentative distances where g is a function we are free to choose[3]. Let L be the largest distance of any removed node. Switch to O-Step.

***O*-Step:** Repeatedly apply the OUT-criterion until no tentative distance is smaller than L. Then switch back to I-Step.

The function $g()$ is chosen in such a way that there is both a constant probability for a large yield in an I-Step and the expected number of subsequent O-Steps is constant. The function $g()$ is chosen dependent of the current phase type. For example, during late intermediate phases we take $g(q) = cq^{2/3}/d^{1/3}$ for some constant c. A super-phase consisting of an I-Step and series of O-Steps is now profitable if at most a constant number of O-Steps is needed and if its total yield is $\Omega(Y_t^{2/3}/d^{1/3})$, highly profitable if its yield is $\Omega((Y_{t/2} - t/2)^{2/3}/(n/t)^{1/3})$. Then one has to show again that a super-phase is (highly) profitable with constant probability.

Theorem 2. INOUT-approach. *Given a random graph from $\mathcal{G}(n, d/n)$ with edge labels uniformly distributed in $[0, 1]$, the SSSP problem can be solved using $r = \mathcal{O}(n^{1/3})$ delete-phases with high probability.*

[3] Note that the implementation does not need to know this function since it uses the faster combined criterion.

4 Parallelization

We now show how the sequential OUT-variant of Sect. 2 can be efficiently implemented on an arbitrary-write CRCW PRAM for random graphs from $\mathcal{G}(n, d/n)$ and random edge weights. The actual number of edges is $m = \Theta(dn)$ whp.

The algorithm keeps a global array tent$(\cdot)$ for all tentative distance values. Each processor P_i, $0 \leq i < p$ is responsible for two sequential priority queues: Q_i and Q_i^*. Each pair (Q_i, Q_i^*) only deals with a subset of nodes, the distribution is made randomly and stored in a global array ind(). Furthermore, each PU maintains a buffer array for incoming requests.

The queues Q_i handle tentative node distances for the nodes they are responsible for, the key of a node $v \in Q_i^*$ is given by $\text{tent}(v) + \delta_o(v)$ where $\delta_o(v) := \min\{c(v,w) : (v,w) \in E\}$; $\delta_o(v)$ is precomputed once and for all upon initialization. The Q_i^* queues are used to efficiently derive the criterion of the OUT-version indicating whether a node can be deleted in a phase. The queues are implemented as relaxed heaps [9] because they provide worst-case running times: `findMin`, `insert` and `decreaseKey` are performed in $\mathcal{O}(1)$ time and `delete`/`deleteMin` in $\mathcal{O}(\log q)$ time where q denotes the local queue size.

Let r be the number of delete-phases which are needed, e.g. for the OUT-variant $r = \mathcal{O}(\sqrt{n})$ whp. For the analysis we fix the number of processors as $p = \max\{\frac{n}{r \log n}, \frac{dn}{r \log^2 n}\}$; so from now on a time bound T implies a work bound pT.

The algorithm works similar to Dijkstra's algorithm: The queues start with only s in $Q_{\text{ind}(s)}$ and $Q^*_{\text{ind}(s)}$ and all other local queues empty. This and the initialization of other arrays and buffers (ind(), outgoing edges, ...) can be done in time $\mathcal{O}((n+m)/p) = \mathcal{O}(r \log^2 n)$ whp, even if the input uses an adjacency-list representation.

While any queue is nonempty the algorithm performs a phase consisting of five steps. These steps are now further explicated together with the most interesting part of their analysis, namely for the case that at most n/r nodes are deleted in this phase.

Step 1 finds the global minimum L of all elements in all Q_i^* and can clearly be performed in $\mathcal{O}(\log p) \leq \mathcal{O}(\log n)$ time.

In **Step** 2 each PU i removes the nodes with $\text{tent}(v) \leq L$ from Q_i and Q_i^*. Let $\breve{R}$ denote the union of all these sets of deleted nodes. Our index distribution ensures that no PU has to deal with more than $\mathcal{O}(\log p + |\breve{R}|/p)$ `deleteMins` whp. A single `deleteMin` or `delete` operation takes $\mathcal{O}(\log n)$ time, thus due to $|\breve{R}| \leq n/r$ and $p = \max\{\frac{n}{r \log n}, \frac{dn}{r \log^2 n}\}$ Step 2 can be performed in $\mathcal{O}(\log^2 n)$ time whp.

In **Step** 3 all PUs cooperate to generate a set $\texttt{Req} := \{(w, \text{tent}(v) + c((v,w))) : v \in \breve{R} \text{ and } (v,w) \in E\}$ of *requests*. By compacting $\breve{R}$ and using prefix sums to schedule the PUs this task can be perfectly load balanced. Since $|\texttt{Req}| = \mathcal{O}\left(d|\breve{R}| + \log n\right)$ whp for $|\breve{R}| \leq n/r$, this step can be performed in time $\mathcal{O}(m/(rp) + \log n) = \mathcal{O}(\log^2 n)$ whp.

Step 4 permutes the requests such that (w, x) is put into a buffer array $B_{\mathrm{ind}(w)}$. Altogether there are at most $\mathcal{O}(d|\breve{R}|)$ requests whp that are spread over p buffers, thus, because of the random node distribution, each buffer gets $\mathcal{O}(\log n + d|\breve{R}|/p) = \mathcal{O}(\log^2 n)$ requests whp (Chernoff bounds, $|\breve{R}| \leq n/r$, $p = \max\{\frac{n}{r \log n}, \frac{dn}{r \log^2 n}\}$). The requests are placed by "randomized dart throwing" [18]. If each processor is responsible for the placement of a group of $\mathcal{O}(\log^2 n)$ requests (which may go to different buffers) Step 4 takes $\mathcal{O}(\log^2 n)$ time whp. The dart throwing progress is regularly monitored. In the unlikely case of stagnation (buffers are chosen too small), the buffer sizes are adapted.

Finally, in **Step** 5 PU i scans buffer i and for each request (w, x) with $x <$ tent(w) it updates tent(w) to x and calls `decreaseKey`(Q_i, w, x), `decreaseKey` $(Q_i^*, w, x + \delta_o(w))$ (respectively `insert` for new nodes). Each operation can be executed in $\mathcal{O}(1)$ time, so for $|\breve{R}| \leq n/r$ Step 5 needs time $\mathcal{O}(\log^2 n)$ whp.

Phases with $|\breve{R}| > n/r$ show whp at least as balanced queue access patterns as those phases deleting less elements, thus time and work of a phase increase at most linearly. Let k_i denote the number of nodes removed in phase i. Then $\sum_{i \leq r} k_i \leq n$. The total time over all phases is $T = \mathcal{O}(\sum_{i \leq r} \lceil k_i r/n \rceil \log^2 n) = \mathcal{O}(r \log^2 n + (nr/n) \log^2 n) = \mathcal{O}(r \log^2 n)$ whp.

For $d > r \log^2 n$ more than n PUs can be used by dropping explicit queues: n global bits denote whether an element is "queued" or not and p/n PUs take care of each buffer area in order to cope with the increased number of requests. Alternatively, one can apply an initial filtering step because all but the $c \log n$ smallest edges per node, c some constant, can be ignored whp without changing the shortest paths [10, 12].

The INOUT-version is supported by p additional priority queues. Initialization of $\delta_i(v) := \min\{c(w, v) : (w, v) \in E\}$ involves collecting the weights of edges that are potentially distributed over $\Omega(d)$ adjacency-lists. For random graphs, the number of incoming edges of $k = \Omega(\log n)$ randomly selected nodes is $\mathcal{O}(dk)$ whp. Thus, we can use the randomized dart throwing to perform the initialization using $\mathcal{O}(dn)$ work whp.

Theorem 3. *If the number of delete-phases is bounded by r then the SSSP can be solved in $\mathcal{O}(r \log^2 n)$ time and $\mathcal{O}(n \log n + m)$ work whp. using* $\max\{\frac{n}{r \log n}, \frac{m}{r \log^2 n}\}$ *processors on a CRCW PRAM.*

The running time can be improved by a factor of $\mathcal{O}(\log n)$ if we choose an alternative implementation for the queues based on the parallel priority queue data structure from [19] which supports `insert` and `deleteMin` for $\mathcal{O}(p)$ elements in time $\mathcal{O}(\log n)$ using p PUs whp. In [7] we show how to augment this data structure so that `decreaseKey` and `delete` are also supported.

A queue is represented by three relaxed heaps: A main heap Q_1, a buffer Q_0 for newly inserted elements plus the $\mathcal{O}(\log n)$ smallest ones and Q_d for elements whose key drops below a bound L' due to a `decreaseKey`. Deleted elements in Q_1 are only marked as deleted. More generally, `delete` and `deleteMin` are most of the time only performed on Q_0 and Q_d and only every $\mathcal{O}(\log n)$ phases a function `cleanUp` is called which guarantees that Q_0 and Q_d do not grow too large. For an analysis we refer to [19, 7].

Corollary 1. *SSSP on random graphs with random edge weights uniformly distributed in* $[0, 1]$ *can be solved on a CRCW PRAM in* $\mathcal{O}(n^{1/3} \log n)$ *time and* $\mathcal{O}(n \log n + m)$ *work whp.*

The approach is relatively easy to adapt to distributed memory machines. The ind-array can be replaced by a hash-function and randomized dart throwing by routing. For random graphs, the PU scheduling for generating requests is unnecessary, if the number of PUs is decreased by a logarithmic factor.

The algorithm can also be adapted to a $\mathcal{O}(n^{1/3+\epsilon})$ time and $\mathcal{O}(n \log n + m)$ work EREW PRAM for an arbitrary small constant $\epsilon > 0$. Concurrent write accesses only occur during the randomized dart throwing. It can be replaced by $1/\epsilon$ reordering phases (essentially radix sorting), such that phase i groups all request for a subset of $p^{1-\epsilon i}$ queue pairs. Processors are rescheduled after each phase. After the last phase all requests to a certain queue pair are grouped together and can be handled sequentially.

5 Adaption to External Memory

The best previous external memory SSSP algorithm is due to [16]. It requires at least n I/Os and hence is unsuitable for large n. For our improved algorithm we use D to denote the number of disks and B to denote the block size. Let r be the number of delete-phases and assume for simplicity that each phase removes n/r elements from the queue.

Furthermore, we assume that $D \log D \leq n/r$ and that the internal memory, S, is large enough to hold *one* bit per node. It is indicated in [7] how to proceed if this reasonable assumption does not hold. We partition the adjacency-lists into blocks of size B and distribute the blocks randomly over the disks. All requests to adjacency-lists of a single phase are first collected in D buffers, in large phases they are possibly written to disk temporarily. At the end of a phase the requests are performed in parallel. If $D \log D \leq n/r$, the n/r adjacency-lists to be considered in a phase will distribute almost evenly over the disks whp, and hence the time spent in reading adjacency-lists is $\mathcal{O}(n/D + m/(DB))$ whp. We use a priority queue without `decreaseKey` operation (e.g. buffer trees [2]) and insert a node as often as it has incoming edges (each edge may give a different tentative distance). When a node is removed for the first time its bit is set. Later values for that node are ignored.

The total I/O complexity for this approach is given by $\mathcal{O}(\frac{n}{D} + \frac{m}{DB} \log_{S/B} \frac{m}{B})$ I/Os whp. The number of disks is restricted by $D = \mathcal{O}(\min\{\frac{n}{r \log n}, \frac{S}{B}\})$.

We note that it is useful to slightly modify the representation of the graph (provide each edge (v, w) with $\delta_o(w)$, the minimum weight of any edge out of w). This allows us to compute the L-value while deleting elements from the queue without the auxiliary queue Q^*. This online computing is possible because the nodes are deleted with increasing distances and the L-value initialized with `findMin()` $+ 1$ can only decrease. The preprocessing to adapt the graph takes $\mathcal{O}(\frac{n+m}{DB} \log_{S/B} \frac{m}{B})$ I/Os.

Theorem 4. *SSSP with r delete-phases can be solved in external memory using $\mathcal{O}(\frac{n}{D} + \frac{m}{DB} \log_{S/B} \frac{m}{B})$ I/Os whp if the number of disks is $D = \mathcal{O}(\min\{\frac{n}{r \log n}, \frac{S}{B}\})$ and S is large enough to hold one bit per node.*

6 Simulations

Simulations of the algorithm have greatly helped to identify the theoretical bounds to be proven. Furthermore, they give information about the involved constant factors.

For the OUT-variant on random graphs with random edge weights we found an average value of $2.5\sqrt{n}$ phases. The refined INOUT-variant needs about $6.0\, n^{1/3}$ phases on the average. A modification of the INOUT-approach which switches between the criteria as described in Sect. 2 takes about $8.5\, n^{1/3}$ phases.

We also ran tests on planar graphs taken from [15, GB_PLANE] where the nodes have coordinates uniformly distributed in a two-dimensional square and edge weights denote the Euclidean distance between respective nodes. The OUT-version finished in about $1.2\, n^{2/3}$ phases; taking random edge weights instead, about $1.7\, n^{2/3}$ phases sufficed on the average. The performance of the INOUT-version is less stable on these graphs; it seems to give only a constant factor improvement over the simpler OUT-variant.

Motivated from the promising results on planar graphs we tested our approach on real-world data: 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 ($n = 157,457$). While repeatedly doubling the number of nodes, the average number of phases (for different starting points) only increased by a factor of about $1.63 \approx 2^{0.7}$; for $n = 157,457$ the simulation needed $6,647$ phases.

7 Conclusions

We have shown how to subdivide Dijkstra's algorithm into delete phases and gave a simple CRCW PRAM algorithm for SSSP on random graphs with random edge weights which has sublinear running time and performs $\mathcal{O}(n \log n + m)$ work whp. Although the bounds only hold with high probability for random graphs, the approach shows good behavior on practically important real-world graph instances.

Future work can tackle the design and performance of more refined criteria for safe node deletions, in particular concerning non-random inputs.

Another promising approach is to relax the requirement of $\mathrm{tent}(v) = \mathrm{dist}(v)$ for deleted nodes. In [7, 17] we also analyze an algorithm which allows these two values to differ by an amount of Δ. While this approach yields more parallelism for random graphs, the safe criteria do not need tuning parameters and can better adapt to inhomogeneous distributions of edge weights over the graph.

Acknowledgements

We would like to thank Volker Priebe for fruitful discussions and suggestions.

References

[1] N. Alon, J. H. Spencer, and P. Erdős. *The Probabilistic Method.* Wiley, 1992.
[2] L. Arge. *Efficient external-memory data structures and applications.* PhD thesis, University of Aarhus, BRICS-DS-96-3, 1996.
[3] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operation. In *11th IPPS*, pages 689–693. IEEE, 1997.
[4] A. Clementi, L. Kučera, and J. D. P. Rolim. A randomized parallel search strategy. In A. Ferreira and J. D. P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, pages 213–227. Kluwer, 1994.
[5] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *26th STOC*, pages 16–26. ACM, 1994.
[6] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.
[7] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. Parallelizing Dijkstra's shortest path algorithm. Technical report, MPI-Informatik, 1998. in preparation.
[8] E. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.
[9] 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.
[10] A. Frieze and G. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Appl. Math.*, 10:57–77, 1985.
[11] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pairs shortest paths in directed graphs. In *4th SPAA*, pages 353–362. ACM, 1992.
[12] R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Math. Oper. Res.*, 10(4):557–564, 1985.
[13] J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
[14] R. M. Karp. The transitive closure of a random digraph. *Rand. Struct. Alg.*, 1, 1990.
[15] D. E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing.* Addison-Wesley, New York, NY, 1993.
[16] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *8th SPDP*, pages 169–177. IEEE, 1996.
[17] U. Meyer and P. Sanders. Δ-stepping: A parallel shortest path algorithm. In *6th ESA*, LNCS. Springer, 1998.
[18] 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.
[19] P. Sanders. Randomized priority queues for fast parallel access. *Journal Parallel and Distributed Computing*, 49:86–97, 1998.
[20] M. Thorup. Undirected single source shortest paths in linear time. In *38th Annual Symposium on Foundations of Computer Science*, pages 12–21. IEEE, 1997.
[21] 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.
[22] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report CS-90-21, Brown University, 1990.