A Parallel Randomized Approximation Scheme for Shortest Paths

Philip N. Klein

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-56
August 23,1991

A parallel randomized approximation scheme for shortest paths*

Philip N. Klein†
Brown University

August 23, 1991

Abstract

We give a randomized parallel algorithm for approximate shortest path computation in an undirected weighted graph. The algorithm is based on a technique used by Ullman and Yannakakis in a parallel algorithm for breadth-first search.

One of the most fundamental and ubiquitous problems in combinatorial optimization is finding shortest paths rooted at one node in a weighted graph. Aside from being important in its own right, the problem arises in algorithms for many other problems, especially those related to flow.

In view of the importance of the single-source shortest paths problem, it is unfortunate that all known *parallel* algorithms for this problem are hopelessly inefficient on sparse graphs. Indeed, the only approach known that achieves significant parallel speed-up involves solving *all-pairs* shortest paths. When the number of processors available is linear in the size of the input graph, this rather brute-force approach yields essentially no speed-up over sequential computation of the shortest path.

*keywords: shortest path, parallel algorithm, approximation algorithm. AMS categories: 68R10, 68Q25.

†Research supported by NSF grant CCR-9012357 and an NSF PYI award. Additional support provided by ONR and DARPA contract N00014-83-K-0146 and ARPA Order No. 6320, Amendment 1

This inability to make efficient use of parallelism in computing shortest paths is of both theoretical and practical significance. A fast and efficient parallel algorithm for this problem remains a major goal for many researchers in parallel graph algorithms.

In this paper, we describe a parallel *approximation* algorithm for computing single-source shortest paths. The algorithm achieves significant speed-up even when only a linear number of processors are available, when the degree of accuracy required is moderate. In the bounds below, we assume the concurrent-write P-RAM as our model of parallel computation. Moreover, we assume ϵ is no more than a constant.

Theorem 0.1 *There is a randomized parallel algorithm that uses $e \log n/\epsilon^{-2}$ processors and time $O(\sqrt{n}\epsilon^{-2} \log n)$ on an undirected graph with e edges and n nodes, to approximate shortest path distances to within a factor of $1+\epsilon$.*

The algorithm is based on a parallel breadth-first search algorithm due to Ullman and Yannakakis [10]. They show that a breadth-first search tree can be constructed in $\tilde{O}(\sqrt{n})$ time using a linear number of processors.

Their algorithm is limited, however, in that it cannot handle weights on nodes or edges. Thus our contribution is to extend their algorithm so that it can do so, although at a slight loss in speed and accuracy. The key technique is the randomized rounding technique of Raghavan and Thompson [7].

Like that of Ullman and Yannakakis, our algorithm can be modified to find $\sqrt{n}$ single-source single-sink approximate shortest paths within the same resource bounds.

Theorem 0.2 *There is a randomized parallel algorithm that uses $e \log n/\epsilon^{-2}$ processors and time $O(\sqrt{n}\epsilon^{-2} \log n)$ on an undirected graph with e edges and n nodes, to find $\sqrt{n}$ single-source single-sink approximate shortest paths. The approximation error is $1+\epsilon$.*

In fact, within the available resources one can afford to find minimum path-sizes for $e \log^2 n$ different source-sink pairs, as long as the number of distinct sources is $O(\sqrt{n})$. We show in the next subsection how the algorithm of Theorem 0.2 could be especially useful in speeding up concurrent flow computation using parallel processing.

Ullman and Yannakakis show that their approach can be used to obtain even faster parallel algorithm, albeit at a cost: the total work done is greater. The same is true for the shortest path approximation algorithm.

Theorem 0.3 *There is a randomized parallel algorithm that uses* $en^{1-2\delta}\log n/\epsilon^{-2}$ *processors and time* $O(n^{\delta}\epsilon^{-2}\log n)$*, to approximate shortest path distances to within a factor of* $1+\epsilon$*, provided* $e \geq n^{2-3\epsilon}$.

0.1 Application to concurrent flow computation

One application for which our algorithm is particularly well-suited is in approximating concurrent flow with uniform capacities. Concurrent flow [9] is an optimization version of multicommodity flow. A special case of concurrent flow, in which all capacities are the same, is the computational bottleneck in implementing two important approximation algorithms, one by Leighton and Rao [6] for finding sparsest cuts in a graph, and one by Raghavan and Thompson [7] for finding a VLSI routing that minimizes channel width.

Klein, Plotkin, Stein, and Tardos [5] have given a fast approximation algorithm for concurrent flow with uniform capacities. The algorithm consists of a sequence of iterations, each essentially a shortest path computation. Consider implementing this algorithm on a parallel machine. A straightforward implementation of Dijkstra's shortest paths in parallel would require $O(n\log n)$ time using $e/n\log n$ processors. Thus using this implementation, one could achieve a speed-up of about $O(e/n)$ over the sequential algorithm. For sparse graphs, then, the speed-up is small.

Using the algorithm of Theorem 0.1, on the other hand, one can compute shortest paths in $O(\sqrt{n}\epsilon^{-2}\log n)$ time using $e\log n/\epsilon^{-2}$ processors. Thus one can achieve a speed-up of $\Omega(e/\sqrt{n})$ in solving concurrent flow to within a constant factor, and consequently in implementing the aforementioned approximation algorithms. This speed-up is $\Omega(\sqrt{n})$ even in the case of sparse graphs.

Admittedly, the time-processor product to achieve this speed-up is larger by a factor of about $\sqrt{n}$. How can we make better use of processors? We use the fact (Theorem 0.2) that the shortest path algorithm can actually find $O(\sqrt{n})$ single-source single-sink shortest paths within the same bounds.

The randomized algorithm of [5] involves a sequence of iterations. In each iteration, a commodity is randomly selected and the shortest path is found from the commodity's source to its sink. If the shortest path is sufficiently short, flow is rerouted. Otherwise, nothing is done.

Recent experiments [4] with a version of the algorithm indicate that the path is sufficiently short only about 8% of the time (on average). Thus

most of the algorithm's time is spent computing shortest paths that are not sufficiently short. By using the algorithm of Theorem 0.2, one can try $O(\sqrt{n})$ paths at a time, and have a much greater chance of finding a sufficiently short path in each iteration. The resulting speed-up should in turn be much greater.

1 Background and overview

A word about terminology in order to forestall confusion. Unless otherwise stated, when we speak of the *length* of a path, we mean the sum of the weights of its edges (rather than the number of its edges). Similarly, *distance* is measured according to edge-weights. The *size* of a path, on the other hand, shall signify the number of nodes in the path, and the *minimum path-size* is the shortest distance measured in number of nodes traversed.

Unless otherwise stated, n and e denote, respectively the number of nodes and the number of edges in the input graph.

1.1 Parallel breadth-first search

The fastest known parallel algorithm for breadth-first search involves repeatedly squaring a matrix, where the elementwise operations are in the min–plus semiring. In fact, this algorithm can compute shortest paths. The technique is well-known; see [1, 2] for more details. The problem with this algorithm is that it requires a great many processors; to achieve $O(\log n)$ time requires about n^3 processors. The processor bound has been improved somewhat [3] in the case of breadth-first search.

The most elementary parallel search technique is parallel breadth-first search, in which the nodes are visited level by level as the search progresses. Level 0 consists of the starting node s, level 1 consists of the neighbors of s, level 2 consists of the neighbors of the neighbors of s (or, rather, those not belonging to previous levels), and so forth.

To go from one level to the next using p processors requires time $O(e/p + 1)$. Initially, the e edges are grouped into p blocks of size roughly e/p, and each block is assigned to a processor. To go from level i to level $i + 1$, each processor inspects the edges in its block; if an edge is *active*—if its tail is in

level i and its head has not yet been traversed—then its head is assigned to level $i+1$.

The problem with this approach is that the time required grows linearly with the number of levels traversed. To keep the time small, we must resort to *limited search*, in which we limit the number of levels traversed. We will use the adjective "k-limited" to signify that only paths consisting of $\leq k$ nodes are under consideration. Thus a *k-limited shortest path from s to t* is a path that is no longer than any path containing $\leq k$ nodes. (We mean to allow the k-limited shortest path to consist of more than k nodes.) Thus, for example, in an n-node graph, an n-limited shortest path is in fact a true shortest path, because every path contains $\leq n$ nodes, whereas an $n/2$-limited shortest path may not be a true shortest path.

1.2 Ullman and Yannakakis's algorithm

The basic version of Ullman and Yannakakis's parallel algorithm for breadth-first search is based on $\sqrt{n}$-limited search. We review a simplified version. We are given a graph and a source s from which to find a breadth-first search tree. First, $O(\sqrt{n}\log n)$ randomly chosen nodes, along with the source, are designated as distinguished. From each of the distinguished nodes x in parallel, p processors conduct a $\sqrt{n}$-limited search, identifying the minimum path-size from x to each node within path-size $\sqrt{n}$. By choosing $p = e/\sqrt{n}$, we can carry out all of these searches in $O(\sqrt{n})$ time using $p\sqrt{n}\log n = e\log n$ processors.

Second, an auxiliary graph is formed on the set of distinguished nodes, where the length of an edge from u to v is defined to be the mininum path-size from u to v found during the limited search. All-pairs shortest paths are computed for the auxiliary graph. This takes total work $O((\sqrt{n})^3 \log n)$, and can easily be done in $O(\sqrt{n})$ time using $e\log n$ processors. In particular, we have the length of the shortest path in the auxiliary graph from the source s to each of the distinguished nodes.

Third, the minimum path-size from the source s to a node v is computed by taking the minimum, over all distinguished nodes x, of the auxiliary-graph shortest path length from s to x plus the minimum path-size from x to v found in x's limited search. One can show that, with high probability, this method yields the minimum path-size for all the nodes v.

The proof is essentially as follows. Fix in advance one minimum-size

path P_v from s to v, for every node v. Now consider the randomly selected distinguished nodes. With high probability, each subpath of P_v of size $\sqrt{n}$ contains a distinguished node. Thus the path P_v is broken up into subpaths starting and ending with distinguished nodes (except for the last subpath, which ends on v), such that each subpath has size at most $\sqrt{n}$. These subpaths are traversed in the limited search. Hence the minimum path-size from s to the last distinguished node of P_v is correctly computed in the all-pairs computation, and the minimum path-size from s to v is correctly computed in the last step of the algorithm.

Given $\sqrt{n}$ source-sink pairs, one can use essentially the same algorithm to find corresponding minimum path-sizes. Let the set of distinguished nodes include the $\sqrt{n}$ sources in addition to the randomly selected nodes. The first and second steps are otherwise the same. In the third step, for each of the given source-sink pairs (s, t) in parallel, one computes the minimum path-size from s to t, as follows. The path-size is the minimum, over all distinguished nodes x, of the auxiliary-graph shortest path length from s to x plus the minimum-path size from x to t found in x's limited search. Computing that minimum for a single source-sink pair takes $O(\sqrt{n})$ work, so the total work for the third step is only $O(n)$. Indeed, within the available resources one can afford to find minimum path-sizes for $e \log n$ different source-sink pairs, as long as the number of distinct sources is $O(\sqrt{n})$.

Ullman and Yannakakis's approach does not directly work with weighted graphs because there is no apparent way to find even the $\sqrt{n}$-limited shortest paths within the available resource bounds. Our contribution is a method that approximates these shortest paths.

We provide a procedure that will find $\sqrt{n}$-limited *approximate* shortest paths in $O(\sqrt{n}\epsilon^{-2} \log n)$ time using $e/(\sqrt{n}\epsilon^{-2})$ processors. The estimated lengths are within a factor of $1 + \epsilon$ of correct.

Note that this procedure is sufficient to make Ullman and Yannakakis's procedure work for weighted graphs; the second and third steps don't depend on the weights being one. The resulting algorithm, though slower by a factor of $O(\epsilon^{-2} \log n)$, performs only a factor of $O(\log n)$ more work.

1.3 Weighted parallel breadth-first search

There is an obvious approach to extending parallel breadth-first search to handle positive integral weights. *Weighted parallel breadth-first search* is just

like ordinary parallel breadth-first search, except that for each active edge, if the edge's weight is one, its head is assigned to the next level, and if the edge's weight is more than one, its weight is decreased by one. Thus level i consists of those nodes at distance i from the source. Assuming the graph is undirected, we can also allow zero weights; in a preprocessing step, the processors contract all zero-weight edges. This step does not change shortest-path lengths.

Weighted parallel breadth-first search has the same disadvantage as ordinary parallel breadth-first search, only more so. The time required grows linearly with the distance traversed. This is especially a problem when the weights are large. To alleviate this disadvantage, the obvious fix is to uniformly shrink all weights. This in itself is not enough; the search crucially depends on the weights being integral. Thus the subtlety is in rounding the shrunken weights to integers without substantially changing the length of the shortest path.

1.4 Our randomized approach to approximating shortest paths

There is a well-established technique for rounding weights without changing sums of weights too much; the randomized rounding technique of Raghavan and Thompson, originally proposed for VLSI routing. The idea is that a nonintegral value is rounded up or down, with the decision being made randomly; the relative probabilities reflect how close the value is to the next higher and next lower integer. This is the technique we use in our weighted limited search procedure.

In Sections 2 and 3, we outline an procedure, FULLSEARCH(G, s, k, p, ϵ) for finding approximate k-limited shortest paths from source s in the graph G to within a factor of $1 \pm \epsilon$, where ϵ is a positive number strictly less than one. The procedure uses p processors and takes time

$$O((\frac{|E(G)|}{p} + k\epsilon^{-2}) \log n)$$

When we call this procedure with $k = \sqrt{n}$ and $p = |E(G)|/\sqrt{n}\epsilon^{-2}$, it takes time $O(\sqrt{n}\epsilon^{-2} \log n)$. By using the procedure in Ullman and Yannakakis's algorithm, we prove Theorems 0.1 and 0.2.

2 The procedure SEARCH

In this section, we give a procedure SEARCH$(G, s, d, k, p, \epsilon)$ to perform a k-limited search in a graph G starting at source node s, using p processors. It determines the k-limited approximate shortest path distances from s of those nodes whose distance is at least d and at most $2d$. We show that estimated path length will with high probability be accurate to within a relative error of ϵ.

By calling SEARCH$(G, s, d, k, p, \epsilon)$ in parallel with $d = 1, 2, 4, \ldots$, one can find the k-limited approximate shortest path distances of all nodes. We show in the next section how to accomplish this with only an $O(\log n)$ factor increase in the number of processors.

Using p processors, the procedure SEARCH$(G, s, d, k, p, \epsilon)$ takes time

$$O(\frac{|E(G)|}{p} + k \log n \epsilon^{-2}) \tag{1}$$

where $|E(G)|$ is the number of edges and n is the number of nodes.

The first step of the procedure is to define the *scale factor* π by

$$\pi = \Theta(k \log n / \epsilon^2 d) \tag{2}$$

Assign randomly rounded scaled edge-lengths

$$\bar{\ell}(e) = \text{round}(\pi \ell(e))$$

where round(x) is a procedure that randomly rounds x to a nearby integer, according to the following rule:

$$\text{round}(x) = \begin{cases} \lceil x \rceil & \text{with probability } x - \lfloor x \rfloor \\ \lfloor x \rfloor & \text{otherwise} \end{cases}$$

Contract all edges whose assigned length $\bar{\ell}(e)$ is zero. since the graph is undirected, this does not affect shortest path lengths. (This is the only place where we use the fact that the graph is undirected.) Then compute shortest paths subject to the new lengths $\bar{\ell}$.

Note that $\bar{\ell}$ is integral, so we can use weighted parallel breadth-first search. Moreover, since we are only interested in path lengths that are at most $2d$, and under scaling such lengths become about $2\pi d$, the number of stages of

search needed is only $O(\pi d)$, which is $O(k \log n \epsilon^{-2})$ by choice of π. Hence the time required is as claimed in (1). The estimated distances can be obtained by unscaling the scaled distances, namely by dividing distances by π.

Theorem 2.1 *With high probability, estimated path length is accurate to within a relative error of* ϵ.

Proof: We want to show that $\pi\ell(P) = \sum_{e \in P} \pi\ell(uv)$ is well-estimated, for each path P under consideration. Write

$$\pi\ell(P) = \sum_{e \in P} (\pi\ell(e) - \lfloor \pi\ell(e) \rfloor) + \sum_{e \in P} \lfloor \pi\ell(e) \rfloor$$

and let b denote the first sum on the right-hand side.

We use versions of Chernoff's bound due to Raghavan [8]: if Ψ is the sum of independent Bernoulli trials, and μ is the expected value of Ψ, then for δ less than twice the base of the natural log, minus one,

$$Pr[\Psi < (1-\delta)\mu] < \exp(-\delta^2\mu/2) \tag{3}$$

and

$$Pr[\Psi > (1+\delta)\mu] > \exp(-\delta^2\mu/4) \tag{4}$$

For each edge e, let $x_e = \text{round}(\pi\ell(v)) - \lfloor \pi\ell(v) \rfloor$. Let $\Psi = \sum_{e \in P} x_e$. Then the expected value of Ψ is b. Moreover, the estimated path length is

$$\bar{\ell}(P) = \pi\ell(P) + \Psi - b$$

The estimated path length differs by too much if and only if

$$|\bar{\ell}(P) - \pi\ell(P)| > \epsilon\pi\ell(P)$$

which holds if and only if

$$|\Psi - b| > \epsilon\pi\ell(P)$$

We use the Chernoff bounds to show that this event is unlikely.

$$\begin{aligned} Pr[|\Psi - b| > \epsilon\pi\ell(P)] &\leq 2\exp(-(\epsilon\pi\ell(P)/b)^2 b/4) \\ &= \exp(-\Omega(k \log n \frac{\pi\ell(P)}{b})) \\ &< \exp(-\Omega(k \log n)) \end{aligned}$$

Thus the probability that path P's estimated length differs by an error exceeding ϵ is at most $exp(-\Omega(k \log n))$, which is at most $1/n^{2k}$ for an appropriate choice of π. We must ensure that none of the paths with at most k nodes have inaccurately estimated lengths. There are at most n^k such paths. The probability that any one has an inaccurate estimated length is at most the sum of the individual probabilities $1/n^{2k}$, which is at most $1/n^k$. Thus with probability at least $1-1/n^k$, the lengths of all such paths are sufficiently accurate. □

3 The procedure FULLSEARCH

The procedure SEARCH$(G, s, d, k, p, \epsilon)$ only accurately estimates the k-limited distances of nodes when those distances are between d and $2d$. To estimate the distances of all nodes, we must call SEARCH in parallel with different values of d. We next show how to do this while only paying a $O(\log n)$ factor in the number of processors.

The key is to use the fact that we're only trying for an estimate of the shortest path lengths to discard some edges for some computations. Namely, if a path contains an edge of weight w, then in computing its length, we can essentially treat as zero all weights less than, say, $\epsilon w/2n$, because the sum of all such weights in the path can be no more than an $\epsilon/2$ fraction of the total weight in the path.

We use this idea in a procedure FULLSEARCH(G, s, k, p, ϵ), which calls SEARCH many times in parallel.

The procedure FULLSEARCH first scans all the edges, and places them in categories according to their weights. For a weight w, let $R(w)$ denote the range $(\epsilon w/4n, w]$. Let $\hat{w}$ be $4n/\epsilon$ times the smallest nonzero weight. The category E_0 consists of those edges with weights in the range $R(2^0\hat{w})$, the category E_1 consists of those edges with weights in the range $R(2^1\hat{w})$, and so on. Note that each edge lies in only $\log_2 2n\epsilon^{-1}$ different categories. We assume ϵ^{-1} is no more than $n^{1/4} \log n$ (else shortest paths are exactly computable by a sequential algorithm in the time alloted), so each edge lies in at most $c \log n$ categories, where c is a small constant.

Second, FULLSEARCH assigns processors to the categories in proportion

to the number of edges in each. For each i for which E_i is nonempty, let

$$p_i = p\frac{|E_i|}{|E(G)|}(1/c\log n)$$

be the number of processors assigned to the category $R(2^s\hat{w})$. Because each edge is in at most $c\log n$ categories, the total number of processors assigned is at most the number p available.

Third, FULLSEARCH creates a graph for each category, containing the edges in that category. Construct the graph G_i from G by contracting all edges with weight at most $(\epsilon/4n)2^i\hat{w}$ and removing all edges with weight greater than $2^i\hat{w}$. If the graph is disconnected, only component that contains the source s is constructed.

Finally, FULLSEARCH calls SEARCH$(G_i, s, d_i, k_i, p_i, \epsilon/2)$ for each i for which E_i is nonempty, where $d_i = \frac{1}{2}(2^i\hat{w})$. These searches take place in parallel; the time for each is

$$O(\frac{|E_i|}{p_i} + k\log n\epsilon^{-2})$$

which, by choice of p_i, is

$$O(\frac{|E(G)|\log n}{p} + k\log n\epsilon^{-2})$$

Call number i estimates the distances in the graph G_i that are between d_i and $2d_i$ to within error $\epsilon/2$. The low-weight edges omitted from G_i could have added at most distance $n(\epsilon/4n)2d_i = (\epsilon/2)d_i$, so the relevant distances in G_i are accurate to within error $\epsilon/2$, compared to the distances in the original graph G. Hence the estimated distances are accurate to within a relative error of ϵ, compared to the original graph distances.

4 Faster (but less processor-efficient) algorithms

Ullman and Yannakakis point out that for a constant $\delta > 0$, one can finds a breadth-first search tree in $\tilde{O}(n^\delta)$ time using $\tilde{O}(en^{1-2\delta})$ processors, provided $e \geq n^{2-3\delta}$. Their algorithm can be adapted to use our limited search procedure FULLSEARCH. We sketch the algorithm.

Let $t = n^\delta$.

1. Pick a set of $O(n \log n/t)$ random distinguished nodes. Add the source node s to the set.

2. Use FULLSEARCH to find t-limited shortest paths from each distinguished node, with error parameter $\epsilon/3$.

3. Let H be the graph whose nodes are the distinguished nodes, where there is an edge between u and v if in step 2 a path between u and v was found. The weight of the edge is taken to be the length of the shortest path found.

4. Select $n \log n/t^2$ superdistinguished nodes among the distinguished nodes.

5. Use FULLSEARCH to find t-limited shortest paths from each distinguished node, with error parameter $\epsilon/3$.

6. Construct the graph J whose nodes are the distinguished nodes, based on the information obtained in step 5, just as H was constructed.

7. Compute all-pairs shortest paths in J.

8. Combine the information obtained from the search in step 2 starting at the source node s with the all-pairs information computed in step 5 to determine approximate shortest path length from s to every superdistinguished node.

9. Combine the information obtained from the previous step with the information from step 5 to determine approximate shortest path length from s to every distinguished node.

10. Combine the information obtained from the previous step with the information from step 2 to determine approximate shortest path length from s to every node.

With high probability, the approximate shortest path lengths computed in step 2 have relative error at most $\epsilon/3$. These approximate lengths are the input lengths for the second search, in step 2. Also, the approximate shortest path lengths computed in step 2 have relative error at most $\epsilon/3$, *relative* to the input lengths. It follows that the relative error relative to the actual lengths is at most ϵ.