A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs *

Surender Baswana†
Max-Planck-Institut für Informatik,
Stuhlsatzenhausweg 85,
66123 Saarbrücken, Germany.
Email : sbaswana@mpi-sb.mpg.de

Sandeep Sen‡
Department of Comp. Sc. and Engg.,
Indian Institute of Technology Delhi,
Hauz Khas, New Delhi-110016, India.
E-mail : ssen@cse.iitd.ernet.in

Abstract

Let $G = (V, E)$ be an undirected weighted graph on $|V| = n$ vertices, and $|E| = m$ edges. A t-spanner of the graph G, for any $t \geq 1$, is a subgraph (V, E_S), $E_S \subseteq E$, such that the distance between any pair of vertices in the subgraph is at most t times the distance between them in the graph G. Computing a t-spanner of minimum size (number of edges) has been a widely studied and well motivated problem in computer science. In this paper we present the first linear time randomized algorithm that computes a t-spanner of a given weighted graph. Moreover, the size of the t-spanner computed essentially matches the worst case lower bound implied by a 43 years old girth conjecture made independently by Erdős [26], Bollobás [19], and Bondy & Simonovits [21].

Our algorithm uses a novel clustering approach that avoids any distance computation altogether. This feature is somewhat surprising since all the previously existing algorithms employ computation of some sort of local or global distance information which involves growing either breadth first search trees up to $\theta(t)$-levels or full shortest path trees on a large fraction of vertices. The truly local approach of our algorithm also leads to equally simple and efficient algorithms for computing spanners in other important computational environments like distributed, parallel, and external memory.

Keywords: Graph algorithms, Randomized algorithms, Shortest path, Spanner

1 Introduction

A spanner is a (sparse) subgraph of a given graph that preserves approximate distance between each pair of vertices. More precisely, a t-spanner of a graph $G = (V, E)$ is a subgraph $(V, E_S), E_S \subseteq E$ such that, for any pair of vertices, their distance in the subgraph is at most t times their distance in the original graph. The parameter t is called the *stretch factor* associated with the t-spanner. The concept of spanners was defined formally by Peleg and Schäffer [29] though the associated notion was used implicitly by Awerbuch [8] in the context of network synchronizers.

*Preliminary version of this work appeared in 30th International Colloquium on Automata, Languages and Programming (ICALP), pages 384-396, 2003.
†Part of the work was done during PhD at I.I.T. Delhi and was supported by a fellowship from Infosys Technologies Ltd., Bangalore.
‡This research was supported in part by an IBM UPP award

The concept of spanner is a beautiful graph theoretic concept in its own right. Moreover, spanners are quite useful in various applications in the area of distributed systems and communication networks. In these applications, spanners appear as the underlying graph structure. In order to build compact routing tables [31], many existing routing schemes use the edges of a sparse spanner for routing messages. In distributed systems, spanners play an important role in designing *synchronizers*. A synchronizer, introduced by Awerbuch [8], is a mechanism to simulate a synchronized distributed algorithm in an asynchronous environment. Awerbuch [8], and Peleg and Ullman [30] showed that the quality of a spanner (in terms of stretch factor and the number of spanner edges) is very closely related to the time and communication complexity of any synchronizer for the network. In particular, if there exists a t-spanner of size m' for the network, then a synchronizer can be built that achieves $O(t)$ time complexity and $O(tm')$ communication complexity.

An efficient algorithm for computing a sparse spanner may prove to be useful in efficient computation of all pairs approximate shortest paths also since the running time of most of the shortest path algorithms is proportional to the number of edges in the graph. Running such an algorithm on a sparse spanner would achieve subcubic time at the expense of computing slightly *stretched*, instead of exact, distances. With this simple observation as one of the core ideas, spanners are used implicitly in a number of algorithms for computing all pairs approximate shortest paths [10, 15, 22, 37].

Spanners are also used in computational biology [11] in the process of reconstructing phylogenetic trees from matrices whose entries represent genetic distances among contemporary living species. They are also used in machine embeddings in parallel architecture [18]. For a number of other applications, please refer to the papers [6, 8, 29, 31].

Lower bound on the size of a t-spanner : All the applications of spanners require a t-spanner of smallest possible size (the number of edges). Therefore, from a graph theoretic perspective, the following question arises : How sparse can a t-spanner be ? To answer this question, Peleg and Schäffer [29], and more recently Thorup and Zwick [37] have tried to establish a lower bound on the size of a spanner in terms of its stretch factor. These results use the following simple relationship between the stretch of a spanner and the girth (length of the smallest cycle) of a graph.

A graph has girth at least $t+2$ *if and only if it does not have a* t*-spanner other than the graph itself.*

A classical result from graph theory shows that every graph with $n^{1+1/k}$ edges must have a cycle of length at most $2k$. (Alon *et al.* [3] show that even $\frac{1}{2}n^{1+1/k}$ edges are in fact enough). It has been conjectured by Erdős [26], Bollobás [19], and Bondy and Simonovits [21] that this bound is indeed tight. Namely, for any $k \geq 1$, there are graphs with $\Omega(n^{1+1/k})$ edges that have girth greater than $2k$. However, the proof exists only for the cases $k = 1, 2, 3$ and 5. Since any graph has a bipartite subgraph with at least half the edges, the conjecture implies the existence of graphs with $\Omega(n^{1+1/k})$ edges and girth at least $2k + 2$. These graphs can't have any t-spanner for $t < 2k + 1$, except the graph itself. This establishes a lower bound of $\Omega(n^{1+1/k})$ on the worst case size of a spanner with stretch $2k$ or $2k - 1$.

1.1 An overview of the existing techniques and algorithms

The distance between any two vertices in not merely a function of the edges in their local neighborhood. However, the task of selecting a sparse set of edges that approximates all pairs distances can be achieved by ensuring a proposition which is somewhat local. Suppose we have a subset $E_S \subset E$ that ensures the following proposition for every edge $(x, y) \in E \backslash E_S$.

$\mathcal{P}_t(x, y)$: the vertices x and y are connected in the subgraph (V, E_S) by a path consisting of at most t edges, and the weight of each edge on this path is not more than that of edge (x, y).

Consider any pair of vertices $u, v \in V$, and the shortest path Π_{uv} between the two in the graph $G = (V, E)$. It follows that, each edge e on this path, that is missing in the subgraph (V, E_S), is stretched by a factor at most t. Applying this argument for each missing edge on the path Π_{uv}, it follows that the shortest path is stretched in the subgraph by factor t at most. In other words, (V, E_S) is a t-spanner of G. A number of existing algorithms [6, 10, 22] in fact, to compute a t-spanner, are based on this approach of ensuring $\mathcal{P}_t$ for each missing edge.

Althöfer *et al.* [6] gave the first algorithm for computing a t-spanner for weighted graphs. Their algorithm is similar to Kruskal's algorithm for computing a minimum spanning tree. The edges of the graph are processed in the increasing order of their weights. To begin with, the spanner $E_S = \emptyset$ and the algorithm adds edges to it gradually. The decision as to whether an edge, say (u, v) has to be added (or not) to E_S is made as follows:

If the distance between u and v in the subgraph induced by the current spanner edges E_S is more than $t \cdot weight(u, v)$, then select and add the edge to E_S, otherwise discard the edge.

It follows that $\mathcal{P}_t(x, y)$ would hold for each edge missing in E_S, and so at the end of the process, the subgraph (V, E_S) will be a t-spanner. Moreover, the girth of the graph (V, E_S) is at least $t + 1$. Note that a graph with more than $n^{1+1/k}$ edges must have a cycle of at most $2k$ edges (see [3]). Hence for $t = 2k - 1$, the above algorithm computes a $(2k - 1)$-spanner of size $O(n^{1+1/k})$, which is indeed optimal based on the lower bound mentioned earlier. A simple $O(mn^{1+1/k})$ implementation of the algorithm follows easily. Recently Roditty and Zwick [35] gave an $O(kn^{2+1/k})$ time implementation which is based on a simple algorithm for incrementally maintaining a single source shortest paths tree up to a given distance.

Algorithms of computing spanners have appeared implicitly in the preprocessing phase of a number of data structures for computing approximate shortest paths [10, 22, 37]. The motivation behind these data structures is to store all pairs approximate distances compactly in subquadratic space and answer any approximate distance query efficiently. These data structures include *neighborhood covers* by Awerbuch *et al.* [10], *pairwise covers* by Cohen [22], and *approximate distance oracles* by Thorup and Zwick [37]. In fact spanner lies implicitly at the core of all these data structures, and these algorithms compute spanners as a byproduct. The algorithms of Awerbuch *et al.* [10] and Cohen [22] employ construction of breadth first search (BFS) trees up to level $\geq k$ from a fraction of vertices. Cohen's algorithm requires expected $O(mn^{1/k})$ time to produce a spanner with size $O(kn^{1+1/k})$ and stretch $(2k + \epsilon)$, which is a bit larger than the optimal $(2k - 1)$ stretch. The algorithm of Awerbuch *et al.* achieves stretch $64k$ which is even larger. The fastest known algorithm for computing a $(2k - 1)$-spanner with essentially optimal size-stretch trade off is by Thorup and Zwick [37]. Their algorithm computes a $(2k - 1)$-spanner of size $O(kn^{1+1/k})$, and its expected running time is $O(kmn^{1/k})$. Their algorithm employs construction of full shortest paths trees from $O(n^{1/k})$ vertices.

All the previously existing algorithms for computing a $(2k - 1)$-spanner involve local or global distance computation : building either BFS trees up to level $\geq k$ or full shortest paths trees from a fraction of vertices. In fact, it seems quite natural also, at least at first glance, that the task of computing a spanner - "*selecting a sparse set of edges that approximates the pairwise distances*" might require some sort of distance computation. However, note that there is a worst case $\Omega(m)$ bound on the best known algorithm for computing just a single shortest paths tree or a k-level BFS tree for any $k > 1$. Therefore, pursuing any such approach that involves distance computation can not lead to a linear time algorithm for computing a $(2k - 1)$-spanner.

1.2 Our contribution

Perhaps surprisingly, we prove that a $(2k-1)$-spanner of essentially optimal size can be computed without any sort of (local or global) distance computation, and that too in just linear time. To achieve this goal, we employ a novel clustering approach in order to ensure the proposition $\mathcal{P}_t$ for each non-spanner edge. The main result (c.f. Theorem 4.3) of this paper is the following :

> Given a weighted graph $G = (V, E)$, and integer $k > 1$, a spanner of $(2k-1)$-stretch and $O(kn^{1+1/k})$ size can be computed in expected $O(km)$ time.

The simplicity of the algorithm can be judged from the fact that our algorithm for computing a $(2k-1)$-spanner executes $O(k)$ rounds, and in each round it essentially explores adjacency list of each vertex to prune dispensable edges. This extremely local approach is so useful that our algorithm can be adapted very easily in various other computational environments with arguably optimal performance as follows.

- In synchronous distributed model, a $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size can be computed in $O(k^2)$ rounds and the total communication complexity will be $O(km)$ (see Theorem 5.1). Thus, the time complexity and communication complexity are away from optimal by a factor of at most k^2 and k respectively.

- In the external memory model, a $(2k-1)$-spanner of $O(kn^{1+1/k})$ size can be computed essentially in the same (expected) time as that of sorting m integers in external memory (see Theorem 5.2). Unlike having a linear time complexity in RAM model, integer sorting in external memory has same lower and upper bound as that of general sorting [1]. Needless to say, sorting is one of the most primitive tasks in external memory.

- In CRCW PRAM model, a $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size can be computed with optimal speed-up in $O(k\tau)$ steps for any $\tau \geq \log^* n$ (see Theorem 5.4). The algorithm employs primitive parallel subroutines like computing the smallest element, semisorting and multiset hashing.

With a little variation of our 3-spanner algorithm, one gets a *parameterized* 3-spanner, which plays a crucial role in improving the running time of existing algorithms for all pairs approximate shortest paths problem [15, 14]. A parameterized 3-spanner defined for a graph $G = (V, E)$ and a subset $S \subset V$ (as a parameter) is a 3-spanner with the additional feature that it preserves all those paths whose vertices are neither adjacent to nor members of the set S. It is this unique feature of "achieving sparseness while preserving some essential distances" that it is employed in constructing approximate distance oracles (introduced by Thorup and Zwick [37]) in quadratic time [15].

1.3 Other related work

The notion of a spanner has been generalized in the past by many researchers. We present a brief description of this work below.

Additive spanners : A t-spanner as defined above approximates pairwise distances with multiplicative error, and can be called a multiplicative spanner. In an analogous manner, one can define spanners that approximate pairwise distances with additive error. Such a spanner is called an additive spanner and the corresponding error is called *surplus*. However, very little is done in the area of additive spanners.

Aingworth *et al.* [2] presented the first additive spanner of size $O(n^{3/2} \log n)$ with surplus 2, and the construction was slightly improved by Dor *et al.* [24], and Elkin and Peleg [25]. Baswana *et al.* [16] presented a construction of $O(n^{4/3})$ size additive spanner with surplus 6. It is a major open problem if there exists any sparser additive spanner.

(α, β)-spanner : Elkin and Peleg [25] introduced the notion of (α, β)-spanner for unweighted graphs, which can be viewed as a hybrid of multiplicative and additive spanners. An (α, β)-spanner is a subgraph such that the distance between any pair of vertices $u, v \in V$ in this subgraph is bounded by $\alpha\delta(u, v) + \beta$, where $\delta(u, v)$ is the distance between u and v in the original graph. Elkin and Peleg showed that an $(1 + \epsilon, \beta)$-spanner of size $O(\beta n^{1+\delta})$, for arbitrarily small $\epsilon, \delta > 0$, can be computed at the expense of sufficiently large surplus β. The surplus, though independent of n, depends quite heavily on ϵ and β. In particular, $\beta(\epsilon, \delta) = 2^{(\log 1/\delta - 1)(\log\log 1/\delta + \log 1/\epsilon)}$. Recently Thorup and Zwick [38] introduced a spanner where the additive error is sublinear in terms of the *distance* being approximated. They show that for an unweighted graph, there exists a spanner of size $O(kn^{1+1/k})$ such that for any pair of vertices $u, v \in V$, if $\delta(u, v) = d$, then the distance between them in the spanner is at most $d + O(d^{1-\frac{1}{k-1}})$.

Distance Preservers : Another graph object similar to spanner is the *distance preserver*, which has been recently introduced by Bollobás *et al.* [20]. A subgraph is said to be a d-preserver if it preserves exact distances for each pair of vertices which are separated by distance at least d. Efficient construction of d-preservers has been presented in [20, 23].

Light-weight spanners : In some applications of the spanner, there is a cost factor associated with each edge, which is equal to its weight. For such applications, it is essential to compute a spanner with very few edges and very small total edge weight. A *lightness* parameter is defined for a subgraph as the ratio of total weight of all its edges and the weight of the minimum spanning tree of the graph. Awerbuch *et al.* [9] showed that for any weighted graph and integer $k > 1$, there exists a polynomially constructible $O(k)$-spanner with $O(k\rho n^{1+1/k})$ edges and $O(k\rho n^{1/k})$ lightness, where $\rho = \log(Diameter)$.

In addition to the above work on the generalization of spanners, a lot of work has also been done on computing spanners for special classes of graphs, e.g., chordal graphs, unweighted graphs, and Euclidean graphs. For chordal graphs, Peleg and Schäffer [29] designed an algorithm that computes a 2-spanner of size $O(n^{3/2})$, and a 3-spanner of size $O(n \log n)$. For unweighted graphs, Halperin and Zwick [27] gave an $O(m)$ time algorithm to compute a $(2k - 1)$-spanner of $O(n^{1+1/k})$ size. Salowe [36] presented an algorithm for computing a $(1 + \epsilon)$-spanner of a d-dimensional complete Euclidean graph in $O(n \log n + \frac{n}{\epsilon^d})$ time. However, none of the algorithms for these special classes of graphs seem to extend to general weighted undirected graphs.

1.4 Organization of the paper

The paper has been organized as follows. In the following section, as a warm-up, we present an $O(m)$ expected time algorithm for computing a 3-spanner, and expose some of the key ideas (clustering of vertices) that we formalize and extend in section 3. We present our linear time sequential algorithm for computing $(2k - 1)$-spanner in section 4. We outline the distributed, external memory, and parallel algorithms for computing a $(2k - 1)$-spanner in section 5.

Throughout the paper, unless stated otherwise, we assume that the undirected graph has the aug-

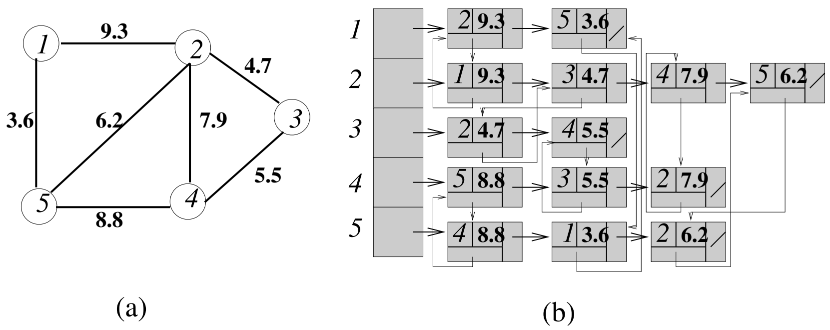

Figure 1: (a) an undirected weighted graph (b) an augmented adjacency list representation

mented adjacency lists representation, wherein for each edge (u, v), the two nodes associated with the edge (in the adjacency lists of each of u and v) have addresses of each other. See Figure 1 given below.

This representation will be helpful in the following way. The graph is undirected and therefore, an edge, say (u, v) appears twice : once each in the adjacency lists of u and v. While processing vertex u, if we decide to delete an edge (u, v) from the graph, we have to delete the edge from the adjacency list of vertex v too. The above representation makes it possible to perform this operation in constant time.

If the initial graph has simple adjacency lists representation, we can get its augmented adjacency lists representation in $O(m)$ processing time. Without loss of generality, it is also assumed that all edge weights are distinct.

2 Computing a 3-spanner

In order to compute a 3-spanner of a given weighted graph $G = (V, E)$, the objective is to select $O(n^{3/2})$ edges to be included in the spanner out of (potentially $\theta(n^2)$) edges E of the graph, and still ensure that the distance between any pair of vertices in the spanner is not more than three times their actual distance. To meet the size constraint of a 3-spanner a vertex, on an average, should contribute $\sqrt{n}$ edges to the spanner. So the vertices with degree $O(\sqrt{n})$ are easy to handle since we can select all their edges in the spanner. The vertices with higher degree pose the following problem : *which $O(\sqrt{n})$ edges should be chosen out of potentially $\theta(n)$ edges incident on a (high degree) vertex ?* Our algorithm employs a novel clustering scheme for such vertices. To begin with, we have a set of edges E' initialized to E, and empty spanner E_S. The algorithm processes the edges E', moves some of them to the spanner E_S and discards the remaining ones. It does so in the following two phases.

1. *Forming the clusters* :
   We choose a sample $\mathcal{R} \subset V$ by picking each vertex independently with probability $\frac{1}{\sqrt{n}}$. We form clusters (of vertices) around the sampled vertices. Initially the clusters are $\{\{u\} | u \in \mathcal{R}\}$. Each $u \in \mathcal{R}$ will be referred to as the *center* of its cluster. We process each unsampled vertex $v \in V - \mathcal{R}$ as follows.

   (a) If v is not adjacent to any sampled vertex, we move every edge incident on v to E_S.

   (b) If v is adjacent to one or more sampled vertices, let $\mathcal{N}(v, \mathcal{R})$ be the sampled neighbor that is nearest [1] to v. We move the edge $(v, \mathcal{N}(v, \mathcal{R}))$ to E_S along with every edge that is incident

[1]Ties can be broken arbitrarily. However, it helps conceptually to assume that all weights are distinct

on v with weight less than that of $(v, \mathcal{N}(v, \mathcal{R}))$. The vertex v is added to the cluster centered at $\mathcal{N}(v, \mathcal{R})$.

As a last step of the first phase, we discard all those edges (u, v) from E' where u and v are not sampled and belong to the same cluster.

Let V' be the set of vertices corresponding to the endpoints of the edges E' left after the first phase. It follows that each vertex from V' is either a sampled vertex or adjacent to some sampled vertex, and the step 1(b) has partitioned V' into disjoint clusters each centered around some sampled vertex. Also note that, as a consequence of the last step, each edge of the set E' is an inter-cluster edge. The graph (V', E'), and the corresponding clustering of V' is passed onto the second phase.

2. *Joining vertices with their neighboring clusters* :
   We process each vertex v of graph (V', E') as follows. Let $E'(v, c)$ be the edges from the set E' incident on v from a cluster c. For each cluster c incident to v, we move the least-weight edge from $E'(v, c)$ to E_S and discard the remaining edges.

Let us first bound the number of edges added to the spanner E_S during the algorithm described above. Note that the sample set $\mathcal{R}$ is formed by picking each vertex randomly independently with probability $\frac{1}{\sqrt{n}}$. It thus follows from elementary probability that for each vertex $v \in V$, the expected number of incident edges with weight less than that of $(v, \mathcal{N}(v, \mathcal{R}))$ is at most $\sqrt{n}$. Thus the expected number of edges contributed to the spanner by each vertex in the first phase of the algorithm is at most $\sqrt{n}$. The number of edges added to the spanner in the second phase is $O(n|\mathcal{R}|)$. Since the expected size of the sample $\mathcal{R}$ is $\sqrt{n}$, therefore, the expected number of edges added to the spanner in the second phase is $O(n^{3/2})$. Hence the expected size of the spanner E_S at the end of the algorithm described above is $O(n^{3/2})$. Since we can verify the number of edges added to the spanner, we will repeat the algorithm if it exceeds $2n^{3/2}$; the expected number of repetitions will be $O(1)$ (using Markov's inequality).

We will now show that E_S has the required properties of a 3-spanner. From the description of the first phase of the algorithm, the following Lemma holds.

Lemma 2.1 *If an edge $(u, v) \in E$ is not present in E_S at the end of the first phase, then the weight of edge (u, v) is greater than or equal to the weight of the edge between v and $\mathcal{N}(v, \mathcal{R})$ (the center of the cluster to which v belongs).*

The proximity of vertices of a cluster to its center relative to the external vertices (as mentioned in Lemma 2.1) is used in the following lemma to bound the stretch of the spanner by 3.

Lemma 2.2 *For each edge $(u, v) \in E \backslash E_S$, the assertion $\mathcal{P}_3(u, v)$ holds.*

Proof: It follows from the first phase of the algorithm that u (as well as v) is adjacent to one or more vertices of the sample $\mathcal{R}$, and therefore u (as well as v) belongs to some cluster. There are two cases now.
Case 1 : *(u and v belong to same cluster)*
Let u and v belong to the cluster centered at $x \in \mathcal{R}$ (see Figure 2). It follows from Lemma 2.1 that there is a 2-edge path $u - x - v$ in the spanner with each edge not heavier than the edge (u, v). This provides a justification for discarding all intra-cluster edges at the end of first phase.
Case 2 : *(u and v belong to different clusters)*
Clearly the edge (u, v) was removed from E' during phase 2, and suppose it was removed while processing the vertex u. Let v belong to the cluster centered at $x \in \mathcal{R}$ (see Figure 3).

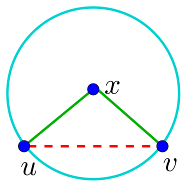

Figure 2: vertex u belongs to the same cluster as the vertex v

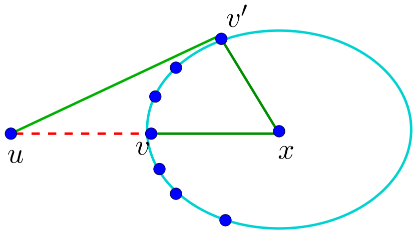

Figure 3: vertex u does not belong to the cluster containing vertex v

In the beginning of the second phase let $(u, v') \in E'$ be the least weight edge among all the edges incident on u from the vertices of the cluster centered at x. So it must be that $weight(u, v') \leq weight(u, v)$. The processing of vertex u during the second phase of our algorithm ensures that the edge (u, v') gets added to E_S. Hence there is a path $\Pi_{uv} = u - v' - x - v$ between u and v in the spanner E_S, and its weight can be bounded as follows.

$$\begin{aligned} weight(\Pi_{uv}) &= weight(u, v') + weight(v', x) + weight(x, v) \\ &\leq weight(u, v') + weight(u, v') + weight(u, v) \quad \{\text{using Lemma 2.1 }\} \\ &\leq 3 \cdot weight(u, v) \quad \{\text{follows from the second phase of the algorithm}\} \end{aligned}$$

□

Using the above lemma, it follows that the spanner (V, E_S) has stretch 3.

Lemma 2.3 *Both phases of the algorithm for computing a 3-spanner can be executed in $O(m)$ time.*

Proof: Without loss of generality we assume that the vertices are numbered 1 to n. The random sample $\mathcal{R}$ can be chosen in $O(n)$ time. Also the nearest sampled neighbor for a vertex $v \in V$ can be computed by a single traversal of the adjacency list of the vertex v. Let N be the array storing nearest sampled neighbor for each vertex (if exists). The remaining task of the first phase is to select (and add to the spanner), for each vertex v, all the edges incident with weight less than that of the edge $(v, \mathcal{N}(v, \mathcal{R}))$. This can be performed by traversing adjacency list of each vertex.

In the second phase of the algorithm, for each vertex v and a cluster neighboring to v, we have to select the least weight edge between the two. For this purpose, we use an auxiliary array $A[1..n]$ whose entries point to *null* initially. Let $E'(v)$ be the list of edges incident on vertex v in the beginning of the second phase. We scan the list $E'(v)$, and process an edge (v, w) as follows. Let $x \in \mathcal{R}$ be the center of the cluster to which the vertex w belongs (note that the center of the cluster to which vertex w belongs can be accessed in constant time from $N[w]$). If $A[x]$ points to *null*, we shall make $A[x]$ point to the edge (v, w). Otherwise let $A[x]$ already points to some edge, say (v, y). In this case, we shall make $A[x]$ point to the lighter (having less weight) of the two edges (v, y) and (v, w), and discard the other edge from the list $E'(v)$. It is easy to observe that once all the edges incident on v have been processed, the list $E'(v)$ consists of only the least weight edges between v and its neighboring clusters; and these (and only these) edges are stored (through pointers) in array A. Now we perform another scan of this list $E'(v)$, and move each edge (v, w) in the list to E_S, and also make $A[N[w]]$ point to *null*. It can be seen that in this way, just by two traversals of the list $E'(v)$, we can select (and add to the spanner) the least weight edge incident on v from each neighboring clusters of v, and discard other edges. Also note that the array A is restored to its initial state (all its entries pointing to *null*) to be used for another vertex.

Thus with an extra space (arrays A and N) of $O(n)$ size, both the phases of the algorithm for computing a 3-spanner can be implemented in $O(m)$ time.

□

We can thus conclude that for a given weighted undirected graph, a 3-spanner of size $O(n^{3/2})$ can be computed in $O(m)$ expected time.

3 Key ideas underlying the $(2\mathrm{k}-1)$-spanner algorithm

As mentioned in the beginning, the task of computing a $(2k-1)$-spanner for a graph $G = (V, E)$ reduces [2] to finding a subset $E_S \subset E$ such that $\mathcal{P}_{2k-1}(e)$ holds for each edge $e \in E \backslash E_S$. Now, in order to pick such a set E_S of $O(kn^{1+1/k})$ edges from potentially $\theta(n^2)$ edges in a given graph, the key idea is to partition the set of vertices into suitable *clusters*. Recall from the previous section how the clustering of the vertices (by grouping each vertex with its nearest sampled neighbor) proves to be crucial in the computation of a 3-spanner. It was the smaller number of these clusters compared to the number of vertices that helped in getting a bound on the size of the 3-spanner, and it was the proximity of the vertices within a cluster that ensured a bound on the stretch of the spanner.

Our algorithm for computing a $(2k-1)$-spanner employs a *clustering* induced by a set of edges. We now formally define this clustering and a parameter called *radius* of a cluster that captures the proximity of the vertices of the same cluster compared to the vertices outside the cluster.

3.1 Definitions and notations

The following definitions and notations are in the context of a given weighted graph $G = (V, E)$.

Definition 3.1 *A* **cluster** *is a subset of vertices. A* **clustering** *of $V' \subseteq V$ is a partition of V' into clusters. As will soon become clear in the context of our algorithm, each cluster is a singleton set in the beginning, and other vertices are added to the cluster as the algorithm proceeds. We shall denote this (unique) oldest member of a cluster as the* **center** *of the cluster. Formally, a clustering $\mathcal{C}$ can be represented by a function $f_{\mathcal{C}} : V \rightarrow V$ such that $f_{\mathcal{C}}(u)$ is the center of the cluster to which the vertex u belongs. Note that $f_{\mathcal{C}}(u) = f_{\mathcal{C}}(v)$ if and only if vertices u and v belong to same cluster. Hence the function $f_{\mathcal{C}}$ associated with the clustering $\mathcal{C}$ can be used to determine whether any two vertices belong to the same cluster or not.*

We now define a clustering induced by a set of edges :

Definition 3.2 *Given a graph $G = (V, E)$, a set of edges $\mathcal{E} \subseteq E$ induces a partition of set V into clusters in the following natural way : two vertices belong to a cluster if they are connected by a path $\Pi \subseteq \mathcal{E}$. (In other words, each connected component is a cluster). We refer to this clustering as the clustering*

[2] Note that this implies a stronger property than required by a spanner.

induced by $\mathcal{E}$. For a cluster c in this clustering, we shall use $\mathcal{E}(c) \subseteq \mathcal{E}$ to denote the edges defining the connected component associated with the cluster c.

Definition 3.3 *Consider a clustering $\mathcal{C}$ induced by some $\mathcal{E} \subseteq E$ in a given graph $G = (V, E)$. The* **radius** *of a cluster $c \in \mathcal{C}$ is the smallest integer r such that the following holds :*

For each edge $(x, y) \in E \backslash \mathcal{E}, x \in c$, there is a path $\Pi \subseteq \mathcal{E}(c)$ from x to $f_{\mathcal{C}}(x)$ of at most r edges each having weight not more than that of the edge (x, y).

Definition 3.4 *A clustering $\mathcal{C}$ induced by $\mathcal{E} \subseteq E$ is a clustering of radius $\leq i$ in the graph $G = (V, E)$ if each of its cluster has radius $\leq i$.*

We shall use the following notations in the rest of the paper.

- $E'(x, c)$: the edges from the set E' that are between the vertices of cluster c and the vertex x.
- $E'(c_1, c_2)$: the edges from the set E' with one endpoint in cluster c_1 and another endpoint in cluster c_2.
- $\min(E')$: the least weight edge from the set E'.

Our algorithm exploits the properties of a clustering of bounded radius as mentioned in the following two Lemmas.

Lemma 3.1 *Let $\mathcal{C}$ be a clustering of radius i induced by $\mathcal{E}$ in a graph $G = (V', \mathcal{E} \cup E')$, and let $c \in \mathcal{C}$ be a cluster. If set $\mathcal{E}$ has been included in the spanner, then for any vertex $u \notin c$, picking the least weight edge from the set $E'(u, c)$ in the spanner will ensure that the proposition $\mathcal{P}_{2i+1}(e)$ holds for each edge $e \in E'(u, c)$.*

Proof: Let the edge (u, y) of weight α be the least-weight edge from the set $E'(u, c)$. Let (u, x) be any

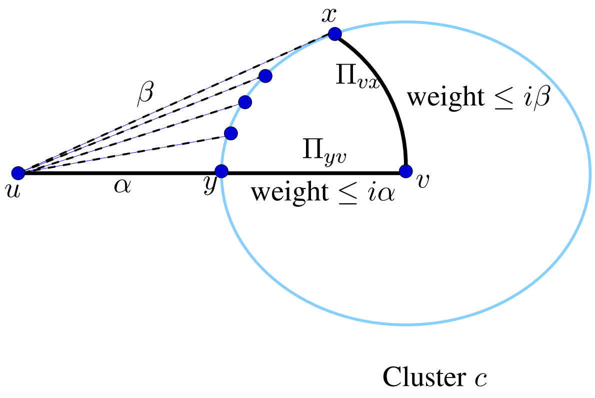

Figure 4: Ensuring that the proposition $\mathcal{P}_{2i+1}$ holds for the set $E'(u, c)$.

other edge of weight $\beta \geq \alpha$ from the set $E'(u, c)$ (see Figure 4). Since the radius of the cluster c is at most i, therefore, there is a path $\Pi_{vx} \subseteq \mathcal{E}(c)$ between vertex x and the center v of the cluster c, and its weight is at most i times β. Using the same argument, we deduce that there is a path $\Pi_{yv} \subseteq \mathcal{E}(c)$ from vertex y to v with weight at most $i\alpha$. Thus there is a path Π_{ux} from vertex u to vertex x formed by concatenating the edge (u, y) and the paths Π_{yv}, Π_{vx} in this order; and its weight can be bounded as follows.

$$\begin{aligned} weight(\Pi_{ux}) &= weight(u, y) + weight(\Pi_{yv}) + weight(\Pi_{vx}) \\ &\leq \alpha + i\alpha + i\beta \leq \beta + i\beta + i\beta \quad \{\text{since } \alpha \leq \beta\} \\ &= (2i+1)\beta \end{aligned}$$

Therefore, we can conclude that if $\mathcal{E}$ has been included in the spanner, then adding the edge (u, y) to the spanner makes the proposition $\mathcal{P}_{2i+1}(e)$ true for each edge $e \in E'(u, c)$. $\square$

Along similar lines we can prove the following Lemma.

Lemma 3.2 *Let $\mathcal{C}$ be a clustering induced by $\mathcal{E}$ in a graph $G = (V', \mathcal{E} \cup E')$, and let $c_1, c_2 \in \mathcal{C}$ be two clusters having radius i and j respectively. If set $\mathcal{E}$ has been included in the spanner, then picking the least weight edge from the set $E'(c_1, c_2)$ in the spanner will ensure that the proposition $\mathcal{P}_{2i+2j+1}$ holds for the entire set $E'(c_1, c_2)$.*

4 Algorithm for computing a $(2\mathbf{k} - 1)$-spanner

4.1 An overview

The algorithm is based on the key observations of a clustering of finite radius mentioned in Lemmas 3.1 and 3.2. It begins with a set E' initialized to E, and empty spanner E_S. The algorithm processes the edges E', moves some of them to E_S and discards the remaining ones. Like the 3-spanner algorithm, it does so in two phases as follows.

The first phase is called *'forming the clusters'* phase and it executes $k-1$ iterations. The ith iteration begins with a clustering of radius $(i-1)$. During the ith iteration, a set of edges from E' are moved to the spanner such that the proposition $\mathcal{P}_{2i-1}$ holds for a possibly large set of edges by Lemma 3.1, which are thus discarded from E'. A new clustering is obtained again for the endpoints of the edges left in E'. In every successive iteration, the expected number of clusters reduces by a factor of $n^{1/k}$ while the radius of clusters increases by at most one unit. At the end of $k-1$ iterations, we obtain a clustering that consists of expected $n^{1-\frac{k-1}{k}} = n^{1/k}$ clusters. This clustering consisting of very *few* clusters and *not-so-large* radius is passed onto the second phase of the algorithm.

The second phase is called *'vertex-cluster joining'* phase. In this phase, each vertex selects the least weight edge from each neighboring cluster and adds it to the spanner (as in the case of 3-spanner).

The algorithm is over at the end of the two phases described briefly above. In another variation called *'cluster-cluster joining'*, we execute only $\lfloor \frac{k}{2} \rfloor$ iterations of the first phase, and then add the least weight edge between each pair of neighboring clusters to the spanner. The *'cluster-cluster joining'* phase employs Lemma 3.2 to ensure that $\mathcal{P}_{2k-1}$ holds for all those edges which are present in the set E' after $\lfloor \frac{k}{2} \rfloor$ iterations but are not selected in the spanner finally. The advantage of this slight variation is the following. For unweighted graphs, it computes a $(2k-1)$-spanner that achieves a stretch strictly better than $(2k-1)$ for any pair of vertices separated by distance larger than one.

4.2 Details of the algorithm

We now describe the details of the two phases of our algorithm for computing a $(2k-1)$-spanner of a weighted graph $G = (V, E)$.

Phase 1 : *Forming the clusters*
This phase executes $k-1$ iterations. The ith iteration begins with tuple $(V', E', E_S, \mathcal{C}_{i-1}, \mathcal{E}_{i-1})$, where E_S is the partially built spanner, E' is the set of edges for which the proposition $\mathcal{P}_{2i-1}$ does not hold yet, V' is the set of endpoints of edges $E' \cup \mathcal{E}_{i-1}$ for some $\mathcal{E}_{i-1} \subseteq E_S$ and $\mathcal{C}_{i-1}$ is a clustering induced by $\mathcal{E}_{i-1}$ in the graph $(V', \mathcal{E}_{i-1} \cup E')$.

Initially, i.e., in the beginning of the first iteration the sets are $E' = E,\ V' = V,\ E_S = \mathcal{E}_0 = \emptyset$, and the clustering $\mathcal{C}_0$ is $\{\{v\} | v \in V\}$.

The ith iteration performs the following four steps in the fixed order.

1. *Forming a sample of clusters* : A sample $\mathcal{R}_i$ of clusters is chosen by picking each cluster from the clustering $\mathcal{C}_{i-1}$ independently with probability $n^{-\frac{1}{k}}$. The set $\mathcal{E}_i$ is initialized to those edges of set $\mathcal{E}_{i-1}$ that define the clusters of $\mathcal{R}_i$. As a consequence, the clustering $\mathcal{C}_i$ is initialized to $\mathcal{R}_i$.

2. *Finding nearest neighboring sampled cluster for each vertex* : For each vertex $v \in V'$ not belonging to any sampled cluster, compute its nearest neighboring cluster (if any) from the set $\mathcal{R}_i$; note that it would be the cluster from $\mathcal{R}_i$ which is incident on v with the lightest edge among all clusters of $\mathcal{R}_i$, and not the cluster with the center at least distance from v. Therefore, it would require each vertex to just scan its adjacency list to compute its nearest neighboring sampled cluster (if any).

3. *Adding edges to the spanner* : To select the spanner edges in the ith iteration, process each vertex $v \in V'$, that does not belong to any sampled cluster, according to the following two cases.

   (a) If v is not adjacent to any sampled cluster, then for each cluster $c \in \mathcal{C}_{i-1}$ adjacent to v, we add the least weight edge from the set $E'(v, c)$ to E_S, and discard the edges $E'(v, c)$ from the set E'.

   (b) If v is adjacent to one or more sampled clusters, let $c \in \mathcal{R}_i$ be the cluster that is adjacent to v with edge, say e_v, of least weight among all the clusters incident on v from the set $\mathcal{R}_i$. We add the edge e_v to the sets E_S and $\mathcal{E}_i$ [3], and discard the entire set $E'(v, c)$ from E'. In addition, we do the following. For each cluster $c' \in \mathcal{C}_{i-1}$ adjacent to vertex v with an edge of weight less than that of e_v, we add the least weight edge from the set $E'(v, c')$ to E_S, and remove $E'(v, c')$ from E'.

   After this 3rd step of the ith iteration, note that the edges remaining in the set E' are only those whose endpoints either belong to or are adjacent to some cluster in $\mathcal{R}_i$. The following crucial observation follows directly from the construction of set $\mathcal{E}_i$ (during steps 1 and 3(b) of the algorithm).

   Observation 4.1 *In the clustering $\mathcal{C}_i$ induced by $\mathcal{E}_i$, each cluster $c \in \mathcal{C}_i$ is the union of a sampled cluster $R \in \mathcal{R}_i$ with the set of all those vertices from V' for whom R was the nearest neighboring sampled cluster in $\mathcal{C}_{i-1}$.*

4. *Removing intra-cluster edges* : All the intra-cluster edges (whose both endpoints belong to the same cluster) of the clustering $\mathcal{C}_i$ are eliminated from E'.

The tuple $(V', E', E_S, \mathcal{C}_i, \mathcal{E}_i)$ at the end of step 4 above is passed onto the $(i+1)$th iteration of the first phase.

Observation 4.1 gives a formal description of the clusterings defined in the successive iterations of the algorithm. The following theorem is the key to understanding of the way the algorithm works, and will also be used for proving its correctness.

[3] This ensures that $\mathcal{E}_i$ is always a subset of the spanner.

Theorem 4.1 *The following assertion holds for each iteration $j \geq 0$.*
$\mathcal{A}(j)$ *: The clustering $\mathcal{C}_j$ induced by the set $\mathcal{E}_j$ in the algorithm is a clustering of radius j in $(V', \mathcal{E}_j \cup E')$.*

Proof: We shall prove the theorem by induction on $j \geq 0$.
Base Case : $j = 0$: In the beginning of the algorithm, $\mathcal{E}_0 = \emptyset$, the clustering is $\mathcal{C}_0 = \{\{v\} | v \in V\}$, and $V' = V, E' = E$. It is easy to observe that each cluster in $\mathcal{C}_0$ is a cluster of radius 0 in the graph $G = (V, E)$. Therefore, the assertion $\mathcal{A}(0)$ holds.
Induction Hypothesis : $j < i$: Let the assertion $\mathcal{A}(i-1)$ hold.
Proof of assertion $\mathcal{A}(i)$:
Recalling the observation 4.1, a cluster $c \in \mathcal{C}_i$ is actually a union $R \cup N_R$, where $R \in \mathcal{R}_i$ and N_R is the set of all those vertices of set V' for whom the cluster R is the nearest neighboring sampled cluster (see Figure 5). The center of the cluster $R \cup N_R$ is the same as that of the cluster R (see Definition 3.1). Since $R \in \mathcal{R}_i \subseteq \mathcal{C}_{i-1}$, it follows from the induction hypothesis that R is a cluster of radius $i-1$ in the

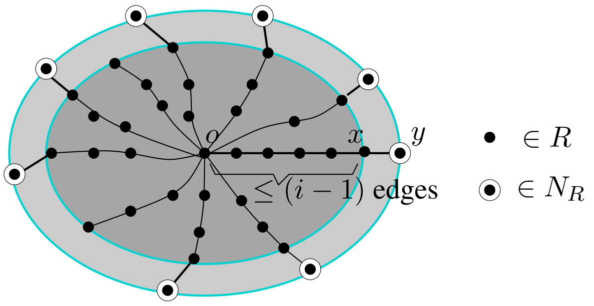

Figure 5: A cluster $c \in \mathcal{C}_i$ as a union $R \cup N_R$

clustering induced by $\mathcal{E}_{i-1}$. That is, for each edge $(x, y) \in E', x \in R$, there is a path $\Pi_{xo} \subseteq \mathcal{E}_{i-1}(R)$ from x to the center o of the cluster R consisting of at most $i-1$ edges each of weight not more than that of (x, y). Since the edges of set $\mathcal{E}_{i-1}(R)$ are present in $\mathcal{E}_i$ too (see step 1 of the algorithm), the radius of cluster R is $i-1$ also in the clustering induced by $\mathcal{E}_i$.

Now consider a vertex $v \in N_R$. During the ith iteration, we add to the set $\mathcal{E}_i$ (and to the spanner), the edge e_v of least weight from the set $E'(v, R)$. Let $u \in R$ be the second endpoint of edge e_v. Therefore, there is a path $\Pi_{vo} \subseteq \mathcal{E}_i$ formed by concatenating e_v with $\Pi_{uo} \subseteq \mathcal{E}_i$ that consists of at most i edges and the weight of each edge on this path is not more than that of e_v (invoke induction hypothesis with $x = u$ and $y = v$). Moreover, as can be noticed from step 3(b) of the algorithm, there is no edge left in the set E' which is incident on v with weight less than that of e_v. Hence the weight of each edge on the path Π_{vo} is not more than that of any edge $(v, z) \in E'$ at the end of the ith iteration. These statements hold for each $v \in N_R$. Hence $R \cup N_R$ is a cluster of radius i in the graph $G = (V', \mathcal{E}_i \cup E')$.

Similar arguments can be given for any other cluster in the clustering $\mathcal{C}_i$. Thus the assertion $\mathcal{A}_i$ holds. Hence by the principle of mathematical induction, the assertion $\mathcal{A}_j$ holds for all $j \geq 0$. $\square$
Using Lemma 3.1 and the theorem given above, we can state the following theorem.

Theorem 4.2 *For each edge $e \in E'$ eliminated from the graph in the first phase, the proposition $\mathcal{P}_{2k-2}$ holds.*

Proof: Let (u, v) be an edge eliminated from E' during the ith iteration. Note that the edges are eliminated from the set E' only in the third or the fourth step of the ith iteration.

Case 1 : (*The edge (u, v) is eliminated from E' during step 3*)
Without loss of generality, assume that the edge (u, v) was eliminated while the vertex u was processed (the case for v is symmetric). Note that we surely add the least weight edge between u and the cluster to which the vertex v belongs. It follows from Theorem 4.1 that each cluster during the ith iteration has radius at most $i - 1$. Therefore, using Lemma 3.1 the proposition $\mathcal{P}_{2i-1}$ holds for the edge (u, v).
Case 2: (*The edge (u, v) is eliminated during step 4*).

In this case both u and v must have been assigned to the same cluster, say $c \in \mathcal{R}_i$. It follows from the step 3 of the ith iteration that the edge (u, v) is at least as heavy as the edge $\min(E'(u, c))$ that we add to the spanner. Moreover, Theorem 4.1 implies that c is a cluster of radius $i - 1$. Therefore, there is a path Π_{uo} (likewise Π_{vo}) from u (likewise v) to the center o of the cluster c consisting of at most i spanner-edges, each of weight not more than that of (u, v). Thus the path formed by concatenating the paths Π_{uo}, Π_{ov} in this order is a path between u and v consisting of at most $2i$ spanner-edges, each of weight no more than that of (u, v). In other words $\mathcal{P}_{2i}$ holds for the edge (u, v).

Since there are $k-1$ iterations in the first phase, it follows that $\mathcal{P}_{2k-2}$ holds for each edge eliminated from the graph in the first phase. □

Lemma 4.1 *The number of edges added to the spanner by the first phase is $O(kn^{1+1/k})$, and its expected running time is $O(km)$.*

Proof: Let v be a vertex belonging to the set V' during the ith iteration of the first phase. All the neighbors of the vertex v are grouped into their respective clusters of the clustering $\mathcal{C}_{i-1}$. Let $c_1, c_2, \cdots, c_l$ be the clusters adjacent to v, and arranged in the increasing order of the weight of their least-weight edge incident on v, i.e., the least weight edge from the set $E'(v, c_j)$ is lighter (has smaller weight) than the least weight edge from the set $E'(v, c_{j+1})$ for all $j < l$.

It follows from the algorithm that for the cluster c_j adjacent to v, we add just one edge (the least weight edge) from the set $E'(v, c_j)$ to the spanner if none of the clusters preceding it, i.e., $c_1, \cdots, c_{j-1}$ are sampled. Since each cluster is sampled independently with probability $n^{-1/k}$, the probability that we add an edge from $E'(v, c_j)$ to the spanner is $(1 - n^{-1/k})^{j-1}$. Thus the expected number of edges contributed to the spanner by a vertex $v \in V'$ is given by

$$\sum_{j=1}^{j=l} \left(1 - n^{-1/k}\right)^{j-1} \leq \frac{1}{n^{-1/k}} = n^{1/k}$$

Thus the expected number of edges added to the spanner in the ith iteration is bounded by $n^{1+1/k}$. We repeat an iteration if the number of edges exceeds $2n^{1+1/k}$; the expected number of repetitions will be $O(1)$ (using Markov's inequality). There are total $k - 1$ iterations in the first phase, so the total number of edges added to the spanner in the first phase is $O(kn^{1+1/k})$.

We now address the running time of the first phase of the algorithm. An iteration of this phase begins with choosing a random sample of clusters and finding the neighboring sampled cluster nearest to each vertex. It is easy to perform these steps in $O(|E'|)$ time. The remaining steps of the iteration (selecting $\min(E'(v, c)$ and/or eliminating $E'(v, c)$) are similar to the second phase of the algorithm for computing a 3-spanner, and thus can be implemented in $O(|E'|)$ time using an extra $O(n)$ size space as follows from Lemma 2.3. Thus the running time of an iteration is $O(|E'|) = O(m)$. As mentioned above, an iteration will be repeated for expected constant number of times to ensure that the number of edges contributed to the spanner in the iteration is of the order of $n^{1+1/k}$. Since there are total $k - 1$ iterations in the first phase, therefore, the expected running time of the first phase is $O(km)$. □

Remark. For an unweighted graph, each vertex would add only a single edge to the spanner in each iteration of the first phase except the iteration in which it is eliminated; during this iteration the expected number of edges that this vertex contributes is at most $n^{1/k}$. Hence the expected number of edges added to the spanner in the first phase is $O(n^{1+1/k} + kn)$ if the graph is unweighted.

Let E' be the set of edges left in the graph at the end of first phase, and let V' be the set of endpoints of edges $E' \cup \mathcal{E}_{k-1}$. We pass the graph (V', E') and the clustering $\mathcal{C}_{k-1}$ of V' to the second phase. Note that $\mathcal{C}_{k-1}$ is a clustering of radius at most $k-1$ in the graph $(V', \mathcal{E}_{k-1} \cup E')$ (see Theorem 4.1).

Phase 2: *Vertex-cluster joining*
The second phase is similar to the second phase of our 3-spanner algorithm, and executes the following step.

- For each vertex $v \in V'$ and each cluster $c \in \mathcal{C}_{k-1}$,
  add the least weight edge from set $E'(v, c)$ to the spanner E_S, and discard $E'(v, c)$ from E'.

For each edge of E' that is not added to the spanner in the second phase, we apply the same argument as that of case 1 in the proof of Theorem 4.2 with $i = k$. Hence the proposition $\mathcal{P}_{2k-1}$ holds for every edge eliminated in the second phase. Using this fact in conjunction with Theorem 4.2, we can conclude that the set E_S at the end of the two phases is a $(2k-1)$-spanner of the given graph $G = (V, E)$.

Since there are $n^{1/k}$ clusters in $\mathcal{C}_{k-1}$, the number of edges added to E_S by the second phase is at most $n^{1+1/k}$. As mentioned above, the execution of this phase is similar to the second phase of the 3-spanner algorithm which takes $O(m)$ time using Lemma 2.3. We have thus proved the following main theorem of this paper.

Theorem 4.3 *Given a weighted graph $G = (V, E)$, and integer $k > 1$, a spanner of stretch $(2k-1)$ and size $O(kn^{1+1/k})$ can be computed in expected $O(km)$ time (for an unweighted graph, the size of the spanner is $O(n^{1+1/k} + kn)$).*

4.3 An alternative to second phase : *Cluster-cluster joining*

There can be a slight variation in the algorithm described in previous subsection that can save a factor of 2 in the number of edges in $(2k-1)$-spanner. This does not give us any asymptotic improvement in the size. However, for unweighted graphs, this variation would ensure a stretch which is strictly less than $2k-1$ for all paths of length more than one. The variation in the algorithm is the following. We don't execute all $k-1$ iterations of the first phase. Instead, we stop after $\lfloor \frac{k}{2} \rfloor$ iterations, and then as an alternative to the second phase, we join each pair of neighboring clusters with the least weight edge between them. The new algorithm is described below.

1. Execute $\lfloor \frac{k}{2} \rfloor$ iterations of the first phase.

   The set $\mathcal{E}_{\lfloor \frac{k}{2} \rfloor}$ of edges partitions the vertices V' into the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ that consists of expected $n^{1-\frac{1}{k}\lfloor \frac{k}{2} \rfloor}$ number of clusters. Moreover, the Theorem 4.1 implies that $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ is a clustering of radius $\lfloor \frac{k}{2} \rfloor$.

   The graph (V', E') with the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ is passed on to the following phase of the algorithm.

2. *Cluster-cluster joining* :

   - If k is odd, then for each pair of clusters $c, c' \in \mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$, we add the least weight edge between the two clusters to the spanner. To execute this, first we merge the adjacency lists of all the vertices belonging to same cluster in the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$. We process the merged list associated with a cluster c as follows. For each cluster $c' \in \mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ incident on c, we select and add the least weight edge from the set $E'(c, c')$ to the spanner.
   - If k is even, then for each pair of neighboring clusters $c \in \mathcal{C}_{\lfloor \frac{k}{2} \rfloor}, c' \in \mathcal{C}_{\lfloor \frac{k}{2} - 1 \rfloor}$, we add the least-weight edge between the two clusters to the spanner. To execute this, first we merge the adjacency lists of all the vertices belonging to same cluster in the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$. For each cluster $c' \in \mathcal{C}_{\lfloor \frac{k}{2} - 1 \rfloor}$ incident on c, we select and add the least weight edge from set $E'(c, c')$ to the spanner.

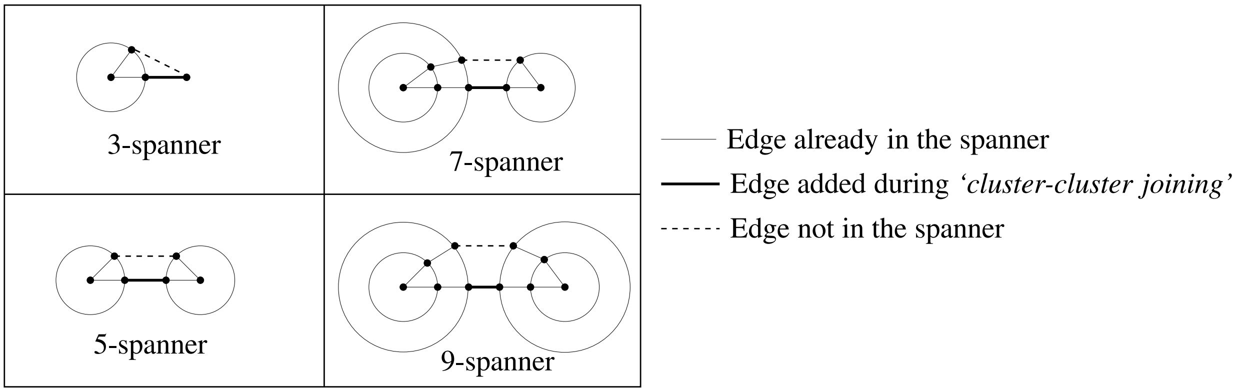

Figure 6: *'cluster-cluster joining'* phase of the new algorithm

Figure 6 shows how the clusters are joined in *'cluster-cluster joining'* phase of our new algorithm for building the spanners of stretch 3,5,7 and 9 (i.e., $k = 2, 3, 4, 5$).

Theorem 4.1 implies that $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ and $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor - 1}$ are clusterings of radius $\lfloor \frac{k}{2} \rfloor$ and $\lfloor \frac{k}{2} \rfloor - 1$ respectively. Therefore it follows from Lemma 3.2 that the proposition $\mathcal{P}_{2k-1}$ holds for each edge of the graph that is processed by *'cluster-cluster joining'* phase but not added to the spanner. So the new algorithm indeed computes a $(2k-1)$-spanner. Also note that for both odd and even cases of k as described above, the processing of a cluster $c \in \mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ is similar to the way we process a vertex in the second phase of the algorithm for 3-spanner (see Section 2). Hence the new algorithm still has an expected running time of $O(km)$.

Lemma 4.2 *The expected number of edges added to the spanner during the* 'cluster-cluster joining' *phase is at most* $n^{1+1/k}$.

Proof: We have to analyze the cases of odd and even k. However, we will deal only with the case of odd k since the case of even k is similar. Let $k = 2\ell + 1$.

Let X_i^v be a random variable which is one if a cluster centered at v appears in $\mathcal{C}_i$, and zero otherwise. Let $\mathcal{N}_i$ be the number of clusters in the clustering $\mathcal{C}_i$. The expected number of edges added to the spanner during the *'cluster-cluster joining'* phase is

$$\sum_{v \in V} \mathbf{Pr}[X_\ell^v = 1] \cdot \mathbf{E}[\mathcal{N}_\ell - 1 | X_\ell^v = 1]$$

As described earlier, the algorithm starts with the clustering $\mathcal{C}_0 = \{\{v\} | v \in V\}$, and a cluster in $\mathcal{C}_i$ survives in $\mathcal{C}_{i+1}$ independently with probability $n^{-1/k}$. It thus follows that $\mathbf{Pr}[X_i^v] = n^{-i/k}$. Owing to the independence used in the sampling of clusters, it also follows that the expected number of clusters in $\mathcal{C}_i$ is bounded by $n^{1-i/k} + 1$ irrespective of whether or not there is a cluster in $\mathcal{C}_i$ which is centered at any particular vertex v. So the expected number of edges added to the spanner during the *'cluster-cluster joining'* phase can be bounded as follows

$$\begin{aligned}\sum_{v \in V} \mathbf{Pr}[X_\ell^v = 1] \cdot \mathbf{E}[\mathcal{N}_\ell - 1 | X_\ell^v = 1] &\leq \sum_{v \in V} \mathbf{Pr}[X_\ell^v = 1] \cdot n^{1-\ell/k} \\ &= n^{1-\ell/k} \sum_{v \in V} n^{-\ell/k} = n^{2-2\ell/k} = n^{1+1/k} \quad \{\text{since } k = 2\ell + 1\}\end{aligned}$$

□

In the following subsection, we shall show that the new algorithm (with *'cluster-cluster joining'* phase) described above would produce spanners with better stretch in case of unweighted graphs.

4.4 Spanners with improved stretch for unweighted graphs

The concept of (α, β)-spanner was introduced by Elkin and Peleg [25] (and briefly mentioned in section 1.3).

Definition 4.3: An (α, β)-spanner of an unweighted graph $G = (V, E)$ is a *subgraph* (V, E_S) such that for all vertices $u, v \in V$, the distance $\delta^*(u, v)$ between them in the spanner is related to their actual distance $\delta(u, v)$ as :

$$\delta^*(u, v) \leq \alpha \cdot \delta(u, v) + \beta$$

It can be seen that a $(2k-1)$-spanner defined earlier is indeed a $(2k-1, 0)$-spanner. In an (α, β)-spanner a single edge may be stretched by as much as $\alpha + \beta$. However the stretch of a long path would be pretty close to α. It is desirable to have smaller multiplicative stretch α at the cost of increased additive stretch β. Now we shall show that a $(2k-1)$-spanner of an unweighted graph computed by our new algorithm described above is indeed a $(\frac{3}{2}k, k-1)$-spanner. We provide the proof for the case when k is odd. However, similar proof can be easily provided for the case when k is even.

First we state the following lemma that follows immediately from *'cluster-cluster joining'* phase of the new algorithm.

Lemma 4.4 *Let $G = (V, E)$ be a given unweighted graphs and (V, E_S) be a $(2k-1)$-spanner as computed by the new algorithm. If (x, y) is an edge, and both x and y belong to clusters $c, c' \in \mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$, then there is a path of length at most k in the spanner between the centers of the clusters c and c'.*

We introduce a notation at this point. For a vertex $x \in V$, $c(x)$ is the vertex x itself if x does not appear in the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$, otherwise $c(x)$ is the center of the cluster containing x in this clustering. In the latter case, there is a path from x to $c(x)$ of length at most $\lfloor \frac{k}{2} \rfloor$.

Lemma 4.5 *Let (u, v) be an edge in a given unweighted graph $G = (V, E)$, and (V, E_S) be a $(2k-1)$-spanner computed by the new algorithm. There is a path between $c(u)$ and $c(v)$ in the spanner (V, E_S) with length at most $k - 2 + \lfloor \frac{k}{2} \rfloor$.*

Proof: If neither u nor v belong to the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ and so $c(u) = u, c(v) = v$, then it must be that the edge (u, v) got eliminated in some iteration $i < \lfloor \frac{k}{2} \rfloor$ of the first phase. So it follows from Lemma 3.1 that there is a path of length at most $k - 2$ between $c(u)$ and $c(v)$ in the spanner. If u as well as v belong to some (same or different) clusters in $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$, then Lemma 4.4 implies that there is a path of length at most k in the spanner that connects $c(u)$ and $c(v)$.

If v belongs to the clustering $\mathcal{C}_{\lfloor \frac{k}{2} \rfloor}$ while u does not belong (or vice versa), then there is a path from $c(u)$ (same as vertex u) to $c(v)$ of length $k - 2 + \lfloor \frac{k}{2} \rfloor$ in the spanner : moving from u to v requires at most $k - 2$ steps using Lemma 3.1, and moving from v to $c(v)$ requires at most $\lfloor \frac{k}{2} \rfloor$ steps. $\square$

Let $u = v_0, v_1, \cdots, v_l = v$ be the shortest path Π_{uv} between u and v in the original graph. Consider the sequence $\langle c(v_0), c(v_1), \cdots, c(v_l) \rangle$. It follows from Lemma 4.5 that there is a path between $c(u)$ and $c(v)$ in the spanner with length $(k - 2 + \lfloor \frac{k}{2} \rfloor)l < \frac{3}{2}kl$. Moreover, moving from u to $c(u)$ (likewise moving from $c(v)$ to v) requires at most $\lfloor \frac{k}{2} \rfloor$ steps in the spanner. Hence, there is a path from u to v in the spanner of length at most $\frac{3}{2}kl + k - 1$. In other words, the spanner is a $(\frac{3}{2}k, k - 1)$-spanner. The size of the spanner is also $O(n^{1+1/k} + kn)$ instead of $O(kn^{1+1/k})$ (refer to the remark stated after the proof of Lemma 4.1).

Theorem 4.4 *An undirected unweighted graph $G = (V, E)$ can be processed in expected linear time to compute a $(\frac{3}{2}k, k - 1)$-spanner of size $O(n^{1+1/k} + kn)$ for any integer $k > 1$.*

Using a couple of new ideas on top of our algorithm, Baswana *et al.* [16] designed an expected $O(km)$ time algorithm that computes a $(k, k-1)$-spanner of size $O(kn^{1+1/k})$ for any unweighted graph. Earlier, Elkin and Peleg [25] had shown that, with higher running time of $O(m\sqrt{n})$, it is possible to compute a $(k - 1, 2k)$-spanner of size $O(kn^{1+1/k})$ for any unweighted graph.

5 Efficient implementation of the algorithm in other computational environments

The unique aspect of our algorithm is its extremely local approach. At any step, processing of a vertex merely involves processing of the edges incident on it, that is, in the immediate neighborhood of the vertex. The reader may note that the earlier algorithms of Awerbuch *et al.* [10] and Cohen [22] involve processing of edges up to level at least k from a vertex, and thus are not local in the true sense. Exploiting the local approach, we design near optimal versions of our algorithm in distributed, external memory, and parallel computational environments.

Recall that since the graph is undirected, an edge between a vertex u and a vertex v appears twice in the adjacency list representation : once in the adjacency list of u, and once in the adjacency lists of v. In each algorithm of this section, we shall treat the two instances of the edge $u - v$, like two different edges : as edge (u, v) while processing the vertex u, and as edge (v, u) while processing the vertex v.

5.1 Implementation in Distributed model

We consider the well known synchronous distributed model of computation. We briefly describe this model as follows. The model consists of a network of nodes and links with some underlying graph $G = (V, E)$ as follows. Each node corresponds to a unique vertex in V (and vice versa), and any two nodes in the network are connected by a link if their corresponding vertices have an edge between them in the set E. Each node has its own local memory and a processor. In this model, the computation takes place in synchronous rounds, where each round involves passing of messages (some data) along links followed by local computation at each node. There are three measures of complexity of any algorithm in this distributed model : number of rounds (time), total number of messages passed along edges (communication complexity), and maximum length of any message passed during the algorithm. Note that the computation performed locally at a node in each round is for free, and not considered as a measure of complexity.

The problem of computing a $(2k-1)$-spanner in synchronous distributed model can be described as follows :

Each link in the distributed network has some positive length (weight) associated with it. The aim is to select $O(kn^{1+1/k})$ edges (links) ensuring a stretch of $(2k-1)$ for any missing link.

We shall now show that our sequential algorithm in RAM model can be adapted in the distributed environment to compute a $(2k-1)$-spanner in $O(k^2)$ rounds and $O(km)$ communication complexity. The expected size of the spanner computed will be $O(kn^{1+1/k})$. Below we present a distributed version of the ith iteration of phase 1 of our sequential algorithm, and show that it will be executed in $O(i)$ rounds with $O(m)$ messages passed.

As a local information each node stores the weight of each of its links. In addition, each node also maintains information about the respective clusters to which it and each of its neighbors belong as the algorithm proceeds. This information is updated through message passing along the links after each iteration.

Distributed algorithm :
The ith iteration begins with the clustering $\mathcal{C}_{i-1}$. The four basic tasks of the ith iteration for computing $(2k-1)$-spanner can be performed in the distributed network as follows.

1. *Forming a sample of clusters* : Center of each cluster $c \in \mathcal{C}_{i-1}$ declares c to be sampled independently with probability $n^{-1/k}$. The center passes this information to those of its neighbors that belong to cluster c. On receiving such message, these neighbors in turn pass the message to their neighbors belonging to cluster c. Since the cluster radius is at most $(i-1)$, it will take $(i-1)$ rounds till each vertex determines whether or not it belongs to a sampled cluster. Also note that the total number of messages passed is $O(m)$.

2. *Finding nearest neighboring sampled clusters for vertices* : Each vertex of a sampled cluster now declares to each of its neighbors that it is now a member of a sampled cluster. Following this, each vertex computes its nearest neighboring sampled cluster.

3. *Adding edges to the spanner* : With the information about its neighbors obtained in step 2 and the local information already present, each vertex selects the edges to be added to the spanner, joins appropriate cluster in the clustering $\mathcal{C}_i$ if needed, and discards unnecessary edges in a way similar to the sequential RAM algorithm.

4. *Removing intra-cluster edges* : Every two neighboring nodes exchange the information about their new cluster in $\mathcal{C}_i$, and discard the link if they belong to same cluster.

It follows that ith iteration gets executed in $O(i)$ rounds and total messages passed in these rounds is $O(m)$. Hence the total number of rounds for the algorithm is $O(k^2)$ and total number of messages communicated is $O(km)$. Also note that each message is of size $O(\log n)$. The spanner computed will have expected $O(kn^{1+1/k})$ edges.

Theorem 5.1 *For any weighted undirected graph, a $(2k-1)$-spanner of expected size $O(kn^{1+1/k})$ can be computed in synchronous distributed environment in $O(k^2)$ rounds and $O(km)$ communication complexity. Moreover, the length of each message communicated is $O(\log n)$.*

5.2 Implementation in external memory

We shall use the external memory model defined by Aggarwal and Vitter [1]. An algorithm for a problem is typically designed with the assumption that the entire data of the problem would reside in the internal memory (RAM). The external memory model is motivated by those applications whose data is too large to be stored completely in the internal memory. In such applications, most of the data resides on the external memory (disk), and only a small portion of it is kept in the internal memory at a time. External memory is slower than internal memory by a factor of 10^6 or even more. Whenever data to be processed is not present in the internal memory, it has to be fetched from the external memory. The data is transferred in units of *blocks*, where a single block can store B words, for some $B > 1$. Let internal memory has a capacity of storing $\mu > 1$ blocks. Block size B, and memory size μ are two parameters of an external memory model. One input/output operation (or simply an I/O) would transfer B *contiguous* words between external memory and internal memory. Executing an algorithm on a huge size application would require a large number of I/O operations. Since each I/O operation is a very time costly operation so that the total time spent in these I/O turns out to be the main bottleneck in the running time of the algorithm. Therefore, the measure of complexity of an algorithm in external memory is the number of I/Os it performs instead of the amount of computation performed in the internal memory. The objective of an efficient external memory algorithm is to minimize the total number of I/O operations during the computation of the solution of some problem. We refer the reader to [28] for an excellent tutorial on external memory algorithms.

Let us consider the task of computing a $(2k-1)$-spanner in external memory. Let a block can store $O(B)$ vertices or edges. If we naively implement our sequential RAM algorithm in external memory, a single iteration would perform $\theta(|E|)$ I/O operations (one I/O for each edge processed). We shall now show that with the set of edges E' arranged in some *suitable* order in the external memory, an iteration of our algorithm would cost $O(|E'|/B)$ I/O operations.

Given two clustering $\mathcal{C}, \mathcal{C}'$ on a set of vertices V', we define an order $\preceq_{(\mathcal{C},\mathcal{C}')}$ on the set of vertices V' and associated edges E' as follows.

- a vertex u would precede vertex v in the order $\preceq_{(\mathcal{C},\mathcal{C}')}$ if
$$f_{\mathcal{C}}(u) < f_{\mathcal{C}}(v) \quad \text{or} \quad f_{\mathcal{C}}(u) = f_{\mathcal{C}}(v) \text{ and } f_{\mathcal{C}'}(u) < f_{\mathcal{C}'}(v)$$
- an edge (u,v) would precede another edge (x,y) in the order $\preceq_{(\mathcal{C},\mathcal{C}')}$ if
$$f_{\mathcal{C}}(u) < f_{\mathcal{C}}(x) \quad \text{or} \quad f_{\mathcal{C}}(u) = f_{\mathcal{C}}(x) \text{ and } f_{\mathcal{C}'}(v) < f_{\mathcal{C}'}(y)$$

If $f_{\mathcal{C}}(u) = f_{\mathcal{C}}(v)$ and $f_{\mathcal{C}'}(u) = f_{\mathcal{C}'}(v)$, it won't matter which of u and v would appear before another in the order $\preceq_{(\mathcal{C},\mathcal{C}')}$. Similarly, if $f_{\mathcal{C}}(u) = f_{\mathcal{C}}(x)$ and $f_{\mathcal{C}'}(v) = f_{\mathcal{C}'}(y)$, then it won't matter which of the two edges (u,v) and (x,y) would appear first in the order $\preceq_{(\mathcal{C},\mathcal{C}')}$. We now state the following Lemma which would highlight the importance of arranging edges according to some order $\preceq_{(\mathcal{C},\mathcal{C}')}$.

Lemma 5.1 *If the list of edges E' is arranged according to the order $\preceq_{(\mathcal{C},\mathcal{C}')}$, then for any two clusters $c \in \mathcal{C}, c' \in \mathcal{C}'$,*

(i) the set of edges $\{(u,v)|u \in c\}$, i.e. the edges emanating from the cluster c appear as a sublist, say L_c.
(ii) the set of edges $E'(c,c')$ appear as a sublist within the sublist L_c.

Data structure :

- The vertices V' are kept in a list. For each vertex $u \in V'$, we store the following additional variables.
  $f_{\mathcal{C}}(u)$: the center of the cluster in clustering $\mathcal{C}$ containing u.
  sampled(u) : a boolean variable which is true during an iteration if u belongs to sampled cluster.
- The data structure for each edge (u,v) has two additional variables : *l-center* storing $f_{\mathcal{C}}(u)$ and *r-center* storing $f_{\mathcal{C}}(v)$.

External memory algorithm :
The four basic tasks of the ith iteration for computing $(2k-1)$-spanner can be performed in external memory as follows.

1. *Forming a sample of clusters* :

   We arrange V' and E' according to the order $\preceq_{(\mathcal{C}_{i-1},\mathcal{C}_0)}$. Consequently, the vertices belonging to same cluster in $\mathcal{C}_{i-1}$ appear together in V'. Scanning the two lists V' and E' simultaneously, we do the following. We pick each cluster for the sample independently with probability $n^{-1/k}$, setting the variable *sampled* for the vertices accordingly, and compute the list E'_R of edges incident on vertices of unsampled clusters from sampled clusters.

2. *Finding nearest neighboring sampled clusters for vertices* :

   We arrange E'_R according to the order $\preceq_{(\mathcal{C}_0,\mathcal{C}_0)}$. As a result, the edges incident on a vertex from all neighboring sampled clusters appear contiguous. We perform a linear scan on this list and among all edges incident on a vertex, we keep only the least weight edge and eliminate all the remaining edges from E'_R.

3. *Adding edges to the spanner* :

   - Add E'_R to the spanner.
   - We arrange the edges E' according to the order $\preceq_{(\mathcal{C}_0,\mathcal{C}_{i-1})}$ so that all the edges incident on a vertex from same cluster in the clustering $\mathcal{C}_{i-1}$ appear contiguous. Selecting the spanner edges (and their deletion from set E'), merely requires a simultaneous scan of the lists V', E'_R, E'.

   Now, to define the clustering $\mathcal{C}_i$, we perform a simultaneous scan on V' and E'_R after arranging them according to $\preceq_{(\mathcal{C}_0,\mathcal{C}_0)}$. We keep only those vertices $u \in V'$ which have either $sampled(u) = true$ or some edge, say $(u,v) \in E'_R$. So each vertex which is neither a member of some sampled cluster nor adjacent to any sampled cluster gets deleted from V'. For an edge $(u,v) \in E'_R$, we need to set $f_{\mathcal{C}}(u) = f_{\mathcal{C}}(v)$ so as to indicate that u is assigned to the cluster containing v in $\mathcal{C}_i$. This can be done by another simultaneous scan of E'_R and V'. The vertices in the final list V' along with their variable $f_{\mathcal{C}}$ constitute the clustering $\mathcal{C}_i$ for the $(i+1)$th iteration.

   For each edge (u,v) left in the set E', we also need to set its variables *l-center* and *r-center* to $f_{\mathcal{C}_i}(u)$ and $f_{\mathcal{C}_i}(v)$ respectively. A simultaneous scan of V' and E' arranged according to $\preceq_{\mathcal{C}_0,\mathcal{C}_0}$

will suffice to set the variable *l-center* of each edge. Now rearranging E' so that the two instances of an edge (for example (u, v) and (v, u)) appear together, we can set the variable *r-center* of each edge by another scan of list E'.

4. *Removing intra-cluster edges* :

   We scan the list of edges E' and delete every edge if its *l-center* is same as its *r-center*.

Remark. While processing a vertex, say u, if we delete an edge (u, v) from the graph, then for consistency we need to delete the edge (v, u) also from the adjacency list of v. In case of RAM model, we could perform this task in constant time by using augmented adjacency list where the nodes associated with the edges (u, v) and (v, u) had pointers to each others. But this would be a costly operation in external memory since a pointer could lead to a block not present in internal memory. This problem can be overcome as follows. While processing the adjacency list of vertex, say u, if we decide to delete an edge say (u, v), we do not delete the edge right away. Instead we mark each such edge that has to be deleted. After 3rd and 4th steps in the iteration, we arrange (sort) the list E' so that the edges with same endpoints appear together in the list. Now, we perform a scan on this list and delete each pair of edges (u, v) and (v, u) if any of them is marked for deletion.

Lemma 5.2 *Each iteration of the algorithm for computing $(2k-1)$-spanner can be executed in external memory in $O(\frac{|E|}{B} \log_\mu \frac{|E|}{B})$ I/O-operations.*

Proof: It follows from the description of the algorithm given above that the ith iteration arranges V' and E' according to the orders $\preceq_{(\mathcal{C}_0, \mathcal{C}_{i-1})}$, $\preceq_{(\mathcal{C}_0, \mathcal{C}_0)}$, $\preceq_{(\mathcal{C}_{i-1}, \mathcal{C}_0)}$ followed by a scan that is performed in I/O-optimal way. Now it is easy to observe that arranging E' or V' according to any of these orders requires integer sort (in fact radix sort) on the labels of the endpoints (and/or *l-center* and *r-center*) of the edges, thus has a running time of $O(|E'|)$ in RAM model. However in external memory, integer sorting has same lower-bound and upper-bound for the running time as that of generic sorting problem, which is $O(\frac{|E'|}{B} \log_\mu \frac{|E'|}{B})$. Thus we conclude that each iteration of our algorithm can be performed in $O(\frac{|E'|}{B} \log_\mu \frac{|E'|}{B})$ I/O-operations. □

Hence we can state the following theorem.

Theorem 5.2 *Given a weighted graph $G = (V, E)$, and an integer $k > 1$, there exists an external memory algorithm for computing a $(2k-1)$-spanner of size $O(kn^{1+1/k})$ that requires expected $O(k\frac{|E|}{B} \log_\mu \frac{|E|}{B})$ I/O-operations.*

5.3 Implementation in parallel environment

The model of computation used by our parallel algorithm is *arbitrary* CRCW PRAM. This model supports concurrent read as well as concurrent write operations into any memory location by multiple processors. In case of simultaneous write operations by two or more processors into a memory location, any of them will succeed. The reader may refer to [33] for a tutorial on parallel algorithms.

We first outline three simple problems whose parallel algorithms will be used in our parallel algorithm for computing spanners.

- *Hashing :* A perfect hash function for a (multi)set X of m integers is an injective function $h : X \to \{1, \ldots, s\}$, where $s = O(m)$, that can be stored in $O(m)$ space and evaluated in constant time by a single processor.

Lemma 5.3 *(Bast and Hagerup [12]) There is a constant $\epsilon > 0$ such that for all $m, \tau \in \mathbb{N}$ with $\tau \geq \log^* m$, simple hashing (or multiset hashing) problem of size m can be solved in CRCW PRAM model using $O(\tau)$ time, $\lceil m/\tau \rceil$ processors, and $O(m)$ time with probability at least $1 - 2^{-(\log m)^{\tau/\log^* m}} - 2^{-m^\epsilon}$ (Las Vegas).*

- *Semisorting :* Given m integers $x_1, \ldots, x_m$ in the range $1..m$, arrange them in an array of size $O(m)$ such that for each $i \in \{1, \ldots, m\}$, all elements of the set $\{j : 1 \leq j \leq m$ and $x_j = i\}$ appear together, separated only by empty cells.

  Lemma 5.4 *(Bast and Hagerup [13]) There is a constant $\epsilon > 0$ such that for all given $m, \tau \in \mathbb{N}$ with $\tau \geq \log^* m$, a semisorting problem of size m can be solved in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors and $O(m)$ space with probability at least $1 - 2^{-m^\epsilon}$ (Las Vegas).*

- *Generalized find min :* Given sets $S_1, S_2, \ldots, S_k$ of real numbers such that $\sum_{i+1}^{k} |S_i| = m$, compute the minimum element of S_i for all i.

  Employing various results from [13], we present a work optimal parallel algorithm that solves generalized find-min problem in $O(\log^* m)$ time with high probability. More precisely,

  Theorem 5.3 *There is a constant $\epsilon' > 0$ such that for all given $\tau, m \in \mathbb{N}$ with $\tau \geq \log^* m$, the generalized find-min problem of size m can be solved on a CRCW PRAM using $O(\tau)$ time, $\lceil m/\tau \rceil$ processors, and $O(m)$ space with probability at least $1 - 2^{-m^{\epsilon'}}$ (Las Vegas).*

  See Appendix for the proof of Theorem 5.3 and the associated algorithm.

Resources (space and processors) : Let τ be any given positive integer with $\tau \geq \log^* m$. We have m/τ processors. Let A be an array storing all the edges. Just like the earlier algorithms in this paper, we shall treat the two instances of an edge (u, v) as two different edges.

Parallel algorithm :

We essentially have to design a parallel algorithm for executing the ith iteration of the first phase of our sequential algorithm (since the execution of the second phase would be identical). The ith iteration executes four tasks. The first task that requires sampling of clusters and the fourth task which requires removing intra-cluster edges can be accomplished deterministically in just constant time with m processors, and in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors. It is mainly the third task of ith iteration that is nontrivial, and we now give a sketch of its efficient implementation in CRCW PRAM. We rearrange the edges E' within array A in such a way that all the sets $E'(v, c), v \in V', c \in \mathcal{C}_{i-1}$ appear as non-overlapping subarrays within A. Recall that similar rearrangement was also carried out in the external memory algorithm (subsection 5.2). To achieve such a rearrangement of edges efficiently in parallel, we proceed as follows. Each edge $(u, v) \in E'$ is assigned a label $(u, f_{\mathcal{C}_{i-1}}(v))$ in the beginning of ith iteration. The desired rearrangement can be achieved by semisorting the edges E' according to these labels. However, the range of these labels could be $1..\theta(n^2)$, which could be much larger than the range $1..O(|E|)$ necessary for carrying out semisorting (see Lemma 5.4). We overcome this problem through multiset hashing. We compute a hash function (using Lemma 5.3) that maps the labels of the edges E' to integers in the range $1..O(|E'|)$, and then we semisort the edges with these new labels. With the edges E' rearranged as mentioned above, the computation of $min(E'(v, c))$ for all $v \in V, c \in \mathcal{C}_{i-1}$ is an instance of the generalized find-min problem which can be solved in $O(\tau)$ time with high probability (see Theorem 5.3). The rearrangement of edges also ensures that the other subtasks like computing the next level clustering $\mathcal{C}_i$, selecting edges to spanner, and discarding dispensable edges from E' can be done in a straightforward way in just $O(\tau)$ time using $\lceil m/\tau \rceil$ processors. Hence the third task can also be accomplished in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors with high probability. The second task requiring computation of nearest sampled neighbor for each vertex can also be accomplished by employing algorithms of semisorting and generalized find-min like in the case of the third task.

Theorem 5.4 *There is a constant $\epsilon > 0$ such that given a weighted graph on n vertices and m edges, an integer k, and $\tau \geq (\log^* n)$, a $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size can be computed in CRCW PRAM model using $O(k\tau)$ time, $O(m)$ space, and $O(m/\tau)$ processors with probability at least $1 - 2^{-m^{\epsilon}}$ (Las Vegas).*

6 Conclusion

We described an expected $O(km)$ time algorithm for computing a $(2k-1)$-spanner of size $O(kn^{1+1/k})$ for any undirected weighted graph. The size is optimal up to a factor of k given the validity of Erdős's *girth conjecture*. Recently Roditty *et al.* [34] derandomized our algorithm while preserving $O(km)$ running time.

The running time of our algorithm as well as the size of the spanner computed are away from their respective worst case lower bounds by a factor of k. For any constant value of k, both these parameters are optimal. However, in the worst case, that is for $k = \log n$, there is deviation by a factor of $\log n$. Is it possible to get rid of this multiplicative factor of k from the running time of the algorithm and/or the size of the spanner computed ? It seems that a more careful analysis coupled with advanced probabilistic tools might be useful in this direction.

For unweighted graphs, we showed that our $(2k-1)$-spanner is indeed a $(\frac{3}{2}k, k-1)$-spanner, i.e., a spanner with reduced multiplicative stretch below $(2k-1)$ at the expense of introducing an additive stretch of $k-1$. Baswana *et al.* [16] extended this approach to design an expected $O(km)$ time algorithm that computes a $(k, k-1)$-spanner of size $O(kn^{1+1/k})$ for any unweighted graph.

The crucial feature of our algorithm is its truly local approach. As a result, our algorithm can be adapted easily in external memory, distributed and parallel computation environment with almost optimal running times. It appears that the local approach might be useful in efficient maintenance of spanners in dynamic graphs as well. Recently Ausiello *et al.* [7] presented a deterministic dynamic algorithm for maintaining spanners with stretch 3 and 5. They essentially dynamize our static algorithm and achieve $O(n)$ update time per edge insertion and deletion. There might be a scope of better update time if the local approach is exploited carefully in conjunction with the randomization.

Acknowledgement

We are thankful to the anonymous referees for their meticulous reading and useful suggestions. We also thank Holger Bast for explaining many details of the paper [13] that helped us in parallelizing our sequential algorithm for $(2k-1)$-spanner.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communication of the ACM*, 31:1116–1127, 1988.

[2] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths(without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

[3] N. Alon, S. Hoory, and N. Linial. The Moore bound for irregular graphs. *Graph Comb. to appear*, 2000.

[4] N. Alon and N. Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. *Journal of Association of Computing Machinery*, 41:422–434, 1994.

[5] N. Alon and J. Spencer. *The probabilistic method.* Wiley, New York, 1992.

[6] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.

[7] G. Ausiello, P. G. Franciosa, and G. F. Italiano. Small stretch spanners on dynamic graphs. In *Proceedings of 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 532–543. Springer, 2005.

[8] B. Awerbuch. Complexity of network synchronization. *Journal of Association of Computing Machinery*, 32(4):804–823, 1985.

[9] B. Awerbuch, A. Baratz, and D. Peleg. Efficient broadcast and light weight spanners. *Tech. Report CS92-22, Weizmann Institute of Science*, 1992.

[10] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhod covers. *SIAM Journal on Computing*, 28:263–277, 1998.

[11] H. J. Bandelt and A. W. M. Dress. Reconstructing the shape of a tree from observed dissimilarity data. *Journal of Advances of Applied Mathematics*, 7:309–343, 1986.

[12] H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *Proceedings of 3rd ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, pages 50–61, 1991.

[13] H. Bast and T. Hagerup. Fast parallel space allocation, estimation and integer sorting. *Information and Computation*, 123:72–110, 1995.

[14] S. Baswana, V. Goyal, and S. Sen. All-pairs nearly 2-approximate shortest paths in $O(n^2 \operatorname{polylog} n)$ time. In *Proceedings of 22nd Annual Symposium on Theoretical Aspect of Computer Science*, volume 3404 of *LNCS*, pages 666–679. Springer, 2005.

[15] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280, 2004.

[16] S. Baswana, K. Telikepalli, K. Mehlhorn, and S. Pettie. New construction of (α, β)-spanners and purely additive spanners. In *Proceedings of 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 672–681, 2005.

[17] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22:221–242, 1993.

[18] S. Bhatt, F. Chung, F. T. Leighton, and A. Rosenberg. Optimal simulations of tree machines. In *In Proceedings of 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 274–282, 1986.

[19] B. Bollobás. *Extremal Graph Theory*. Academic Press, 1978.

[20] B. Bollobás, D. Coppersmith, and M. Elkin. Sparse distance preserves and additive spanners. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 414–423, 2003.

[21] J. A. Bondy and M. Simonovits. Cycles of even length in graphs. *Journal of Combinatorial Theory, Series B*, 16:97–105, 1974.

[22] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[23] D. Coppersmith and M. Elkin. Sparse source-wise and pairwise distance preservers. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 660–669, 2005.

[24] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *Siam Journal on Computing*, 29:1740–1759, 2000.

[25] M. Elkin and D. Peleg. $(1+\epsilon, \beta)$-spanner construction for general graphs. *SIAM Joirnal of Computing*, 33:608–631, 2004.

[26] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.

[27] S. Halperin and U. Zwick. Linear time deterministic algorithm for computing spanners for unweighted graphs. *unpublished manuscript*, 1996.

[28] U. Meyer, P. Sanders, and J. F. Sibeyn. *Algorithms for Memory Hierarchies : Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *LNCS*. Springer, 2003.

[29] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[30] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.

[31] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of Association of Computing Machinery*, 36(3):510–530, 1989.

[32] P. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14:371–380, 1993.

[33] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, California, 1993.

[34] L. Roditty, M. Thorup, and U. Zwick. Deterministic construction of approximate distance oracles and spanners. In *Proceedings of 32nd International Colloquim on Automata, Languagaes and Programming (ICALP)*, volume 3580 of *LNCS*, pages 261–272. Springer, 2005.

[35] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proceedings of 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 580–591. Springer, 2004.

[36] J. D. Salowe. Construction of multidimensional spanner graphs, with application to minimum spanning trees. In *ACM Symposium on Computational Geometry*, pages 256–261, 1991.

[37] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of Association of Computing Machinery*, 52:1–24, 2005.

[38] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proceedings of 17th Annual ACM-SIAM Symposium on Discrete Algorithms (To appear)*, 2006.

Appendix

Proof of Theorem 5.3 :
First we state definitions and lemmas for a few problems which we shall use in our algorithm for generalized find-min problem.

- *Find-min*

  For the standard find-min problem, i.e, computing the minimum of a set of real numbers, the following two lemmas are folklore.

  Lemma 6.1 *([4, 33]) There is a constant $\epsilon > 0$ such that for all given $m, \tau \in \mathbb{N}$, the minimum of m real numbers can be computed on CRCW PRAM in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors with probability at least $1 - 2^{-m^{\epsilon}}$ (Las Vegas).*

  Let c_0 be the constant such that the algorithm from Lemma 6.1, if run for $c_0\tau$ steps would halt with probability $1 - 2^{-(m^{\epsilon})}$. Let *find_min-1* be such an algorithm that would run for at most $c_0\tau$ steps, and if it has not computed the minimum element by that time, it halts and reports the failure.

  Lemma 6.2 *For any constant $\delta > 0$, the minimum of m real numbers can be computed deterministically on a CRCW PRAM in $O(1)$ time using $m^{1+\delta}$ processors.*

  Let *find_min-2* denote the deterministic algorithm from Lemma 6.2.

- *Segmented broadcasting*

  Given m 0-1 numbers $x_1, \ldots, x_m$, compute integers $y_1, \ldots, y_m$ such that $y_i = \max(\{j : 1 \leq j < i \wedge x_j = 1\} \cup \{0\})$.

  Berkman and Vishkin [17] and Ragade [32] showed that segmented broadcasting problem of size m can be solved in $O(\tau)$ time using m/τ processors.

- *Fine-profiling*

  Let $m, h \in \mathbb{N}$ and let $x_1, \ldots, x_m$ be m integers in the range $0..h$. For $i = 1, ..., h$, take $b_i = |\{j : 1 \le j \le m \text{ and } x_j = i\}|$. An h-color fine-profile for $x_1, \ldots, x_m$ is a sequence $\hat{b}_1, \ldots, \hat{b}_h$ of h nonnegative integers such that $b_i \le \hat{b}_i \le cb_i$, for $i = 1, \ldots, h$ and some constant $c > 1$. The h-color fine profiling problem of size m is, given m and h, to compute an h-color fine-profile of m given integers in the range $0..h$.

  Lemma 6.3 (Corollary 10.5 from Bast and Hagerup [13]) *There is a constant $\epsilon > 0$ such that for given $m, \tau \in \mathbb{N}$ with $\tau \ge \log^* m$, m-color fine profiling problems of size m can be solved on CRCW PRAM using $O(\tau)$ time, $\lceil m/\tau \rceil$ processors and $O(m)$ space with probability at least $1 - 2^{-m^\epsilon}$ (Las Vegas).*

- *Processor allocation*

  Definition 6.1 *Given a set of consecutively numbered m processors, and let $x_1, ..., x_j$ be the sizes of requests for processors by j tasks, with $\sum_{i=1}^{j} x_i = m$, processor allocation problem of size m is to allocate m processors to the tasks (x_i processors to ith task) on a CRCW PRAM.*

  Lemma 6.4 (Bast and Hagerup [13]) *There is a constant $\epsilon > 0$ such that for all given $m, \tau \in \mathbb{N}$ with $\tau \ge \log^* m$, processor allocation problem of size m can be solved on a CRCW PRAM using $O(\tau)$ time, $\lceil m/\tau \rceil$ processors and $O(m)$ space with probability at least $1 - 2^{-m^\epsilon}$ (Las Vegas).*

- *Computing sum*

  Lemma 6.5 (Corollary 2.5 from Bast and Hagerup [13]) *For every fixed $\delta > 0$ and for all given integers $m, \tau \ge 2$, the sum of $(\log(m))^{O(1)}$ integers, each of absolute size polynomial in m, can be computed on a CRCW PRAM using $O(\tau)$ time, $\lceil m^\delta/\tau \rceil$ processors and $O(m^\delta)$ space.*

We now use Lemmas 6.3 and 6.5 to solve the following subproblem.

Definition 6.2 *Estimating the sum of squares of small numbers : For any $m, h \in \mathbb{N}$ with $h = (\log m)^{O(1)}$, let $x_1, \ldots, x_m$ be a given sequence of nonnegative integers in the range $0..h$, compute a number R such that*

$$\sum_{i=1}^{m} x_i^2 \le R \le c \sum_{i=1}^{m} x_i^2$$

for some constant $c > 1$.

Let $b_i, 1 \le i \le h$ be the number of occurrences of number i in the sequence. In order to solve the above problem, we first compute the estimates $\hat{b_i}$ with $b_i \le \hat{b}_i \le cb_i$, which, being an instance of *fine-profiling* problem mentioned above, would be solved in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors with high probability (see Lemma 6.3). We then compute the sequence $\{\hat{b}_i \; i^2\}$, followed by the sum $R = \sum_{i=1}^{h} \hat{b}_i \; i^2$ in $O(\tau)$ time using Lemma 6.5. Hence we can state the following Lemma.

Lemma 6.6 *There is a constant $\epsilon > 0$ such that for all given $m, h, \tau \in \mathbb{N}$, with $\tau \ge \log^* m, h = (\log m)^{O(1)}$, the problem of "estimating sum of squares of small numbers" for a sequence of m integers in the range $0..h$ can be solved on CRCW PRAM using $O(\tau)$ time, $\lceil m/\tau \rceil$ processors and $O(m)$ space with probability at least $1 - 2^{-m^\epsilon}$ (Las Vegas).*

For showing high probability bound on the running time of our algorithm, we shall use the following Lemma which is implied by Azuma's inequality [5].

Lemma 6.7 *Let $m \in \mathbb{N}$, let $Z_1, \ldots, Z_k$ be independent random variables with finite ranges, and let S be an arbitrary real function of $Z_1, \ldots, Z_k$ with $\mathbf{E}(S) \geq 0$. If S changes by at most Δ in response to an arbitrary change in a single Z_i, then*

$$\text{For every } z \geq 2\mathbf{E}[S]\,, \qquad \mathbf{Pr}[S \geq z] \leq e^{-z^2/(8\Delta^2 k)}$$

Before we present our algorithm, we would like to address a small implementation issue. We assume that the sets S_i are presented as non-overlapping subarrays in an array A of size $O(m)$. Our algorithm will need to execute some algorithm (*find_min-1* and *find_min-2*) concurrently on each set. This would require allocation of processors to the sets suitably. To achieve this, we partition A into equal sized subarrays and associate ith subarray to ith processor. In case a set spans over many subarrays, each of the its processors has to know the boundaries of the set in A. For this, we first compute the boundaries between different sets in the array A, and then solve an instance of "segmented broadcasting" problem in order to let each processor know the boundaries of the set it will work upon. The entire task will require $O(\tau)$ time using $\lceil m/\tau \rceil$ processors for any $\tau \geq 1$.

Algorithm for *generalized find-min* **problem :**

We shall use the following notation and terminology. A set $S_i, 1 \leq i \leq k$ is an *active* set until we have computed its minimum element, after which it becomes *inactive*. In this terminology, all the sets are active initially. Let $|S_i| = m_i$.

We first present the main idea underlying the algorithm. A naive approach to solve generalized find-min problem would be to employ *find_min-1* algorithm for each set concurrently. The objection against this approach is the following. Probability that *find_min-1* fails for a set could be $\theta(1)$ if the set is of small size. So if there are many small sets, and we keep on repeating *find_min-1* for each active set concurrently, it would require $\theta(\log m)$ repetitions till all sets become inactive, which is much larger than our aim of $O(\log^* m)$ running time. We also can't use directly the *find_min-2* algorithm for generalized find-min since the number of processors required would be much larger than n if there are *large* size sets. To achieve $O(\log^* m)$ time for generalized find-min problem, our algorithm runs in two phases and employs both *find_min-1* and *find_min-2*. In the first phase, we execute the algorithm *find_min-1* for each set concurrently for a certain number of rounds. This would make most of the sets (of large size especially) inactive, and hence the processors initially allocated to these sets can be used by other active sets for executing *find_min-2* algorithm. The second phase executes *find_min-2* algorithm for each remaining active set. Note that an active set S_i would demand $\theta(m_i^2)$ processors so as to employ *find_min-2* algorithm. So we keep on repeating the rounds of *find_min-1* algorithm in phase 1 until the total processor demand (for second phase) of all active sets reduces to $O(m)$, and then execute the second phase.

We now describe the algorithm formally as follows. (In the algorithm, c is the constant from Definition 6.2, $\tau \geq \log^* m$, and a and c' are constants to be fixed later on).

1. (a) Run the algorithm *find_min-1* for each set S_i concurrently until every set, whose size is at least $(\log m)^{a/\epsilon}$, becomes inactive.
   (b) Let I be the set of indices of active sets. Using the algorithm from Lemma 6.6, compute an estimate R such that $\sum_{i \in I} m_i^2 \leq R \leq c \sum_{i \in I} m_i^2$.

While $R > c'm$ do

Execute the algorithm *find_min-1* for each active set.

Recompute R.

2. Allocate m/τ processors such that each active set S_i receives $\theta(m_i^2/\tau)$ processors. Now run the algorithm *find_min-2* for each set separately to compute its minimum element in deterministic $O(\tau)$ time (Lemma 6.2).

Analysis : We shall first show that after a single concurrent execution of *find_min-1* on each active set, the total processor demand $\sum_{i \in I} m_i^2$ of the remaining active sets is $O(m)$ with *high probability*. (Throughout the analysis, for sake of conciseness, we would say that an event happens with *high probability* if it happens with probability at least $1 - 2^{-m^\epsilon}$, for some $\epsilon > 0$).

Given sets $S_1, ..., S_k$, $\sum_i m_i = m$, suppose we execute *find_min-1* algorithm once for each of these sets concurrently. Let X_i be a random variable which is one if set S_i remains active after the execution of *find_min-1* on S_i, and zero otherwise. Note that each of X_i's are independent (this fact will be used later on). The expected processor demand of remaining active sets can be bounded as follows.

$$\begin{aligned}
\mathbf{E}\left[\sum_{i \in I} m_i^2\right] &= \mathbf{E}\left[\sum_{i=1}^{k} X_i . m_i^2\right] = \sum_{i=1}^{k} \mathbf{Pr}[X_i = 1] \cdot m_i^2 \\
&= \sum_{i=1}^{k} 2^{-m_i^\epsilon} m_i^2 \quad \{\text{using definition of } \textit{find_min-1} \text{ after Lemma 6.1 }\} \\
&\leq \sum_{i=1}^{k} c'' \quad \{ \text{ for some constant } c'' \text{ depending upon } \epsilon \} \\
&\leq c''k \leq c''m
\end{aligned}$$

So the expected processor demand of the active sets after a single concurrent run of *find_min-1* would be $O(m)$. Choosing $c' = 2cc''$ in the second step of the algorithm, it follows using Markov's inequality that the expected number of iterations of 'While' loop performed in step 1(b) is $O(1)$. We shall now show, using the method of bounded difference (Lemma 6.7), that the number of iterations is at most 1 with very high probability as follows (the crucial points used are the small sizes of active sets in step 1(b) and the independence of the random variables $X_i, 1 \leq i \leq k$).

Note that each active set in step 1(b) is of size at most $(\log m)^{a/\epsilon}$. As mentioned above, the total processor demand of the active sets left after a round of *find_min-1* is a function of independent random variables X_i, and a change in X_i would change the demand by $\Delta = \Delta(m, \epsilon) = m_i^2 \leq (\log m)^{2a/\epsilon}$. It also follows from the discussion above that the expected processor demand of the existing active sets after each iteration of 'While' loop in step 1(b) is bounded by $c''m$. Hence applying Lemma 6.7, it follows that the processor demand of the remaining active sets is more than $2c''m$ after an iteration of the 'While' loop with probability at most

$$e^{-\theta(m^2)/(8\Delta^2 k)} = e^{-\theta(m)}$$

Thus the step 1(b) of the algorithm will execute at most one iteration of the 'While' loop with high probability.

We can now bound the running time of the entire algorithm quite easily. Choosing an appropriately large value of the constant a, it follows from definition of *find_min-1* (following Lemma 6.1) that step 1(a) would run in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors with high probability. Let us now analyze the step

1(b). In addition to a concurrent run of *find_min-1* which takes $O(\tau)$ time, each iteration of 'While' loop involves computation of R, which would take $O(\tau)$ time using Lemma 6.6 with high probability. From the discussion above, the number of iterations of the 'While' loop is constant with high probability. So the step 1(b) would be executed in $O(\tau)$ time with high probability. The second step involves processor allocation task which can be executed in $O(\tau)$ time with high probability using Lemma 6.4. Another task in the second step is the concurrent execution of algorithm *find_min-2* on each active set which will take deterministic $O(\tau)$ time using m/τ processors (see Lemma 6.2). Hence the second step would also run in $O(\tau)$ time with high probability. We can thus conclude that the algorithm for generalized find-min would run in $O(\tau)$ time using $\lceil m/\tau \rceil$ processors with high probability, that is, with probability at least $1 - 2^{-m^{\epsilon'}}$ for some $\epsilon' > 0$. This concludes the proof of Theorem 5.3.