Constructing the Spanners of Graphs in Parallel

Weifa Liang

Department of Computer Science
Australian National University
Canberra, ACT 0200, Australia
Email:wliang@cs.anu.edu.au

Richard P. Brent

Computer Sciences Lab.
Australian National University
Canberra, ACT 0200, Australia
Email:rpb@cslab.anu.edu.au

Abstract

Given a connected graph $G = (V, E)$ with n vertices, a subgraph G' is an approximate t-spanner of G if, for every u, v $\in V$, the distance between u and v in G' is at most $f(t)$ times longer than the distance in G, where $f(t)$ is a polynomial function of t and $t \leq f(t) < n$. In this paper parallel algorithms for finding approximate t-spanners on both unweighted graphs and weighted graphs with $f(t) = O(t^{k+1})$ and $f(t) = O(Dt^{k+1})$ respectively are given, where D is the maximum edge weight of a minimum spanning tree of G, k is a fixed constant integer, and $1 \leq k \leq \log_t n$. Also, an NC algorithm for finding a $2t$-spanner on a weighted graph G is presented. The algorithms are for a CRCW PRAM model.

1 Introduction

Given a connected graph $G = (V, E)$ with *n* vertices, a subgraph G' is a *t-spanner* (*an approximate t-spanner*) of G if, for every *u*, *v* $\in V$, the distance between u and v in G' is at most t ($f(t)$) times longer than the distance in G, where $f(t)$ is a polynomial function of t and $1 \leq t \leq f(t) < n$. The value of t and $f(t)$ are called the *factors* of G. There are two criteria to measure the sparseness of a spanner, that is, the *size*, defined as the number of edges in the spanner, and the *weight*, defined as the sum of the edge weights in the spanner. The minimum spanning tree (MST) of G is obviously the sparsest spanner in terms of both size and weight, but its factor can be as bad as $n - 1$ [1]. For convenience, we denote by, $wt(MST)$, the sum of the edge weight of the MST. Usually the sparseness of a spanner is judged by comparing it to the size and the weight of the MST.

Much effort has been made in recent several papers [1,6,9,12] regarding spanners on some special graphs such as Euclidean graphs, geometry graphs and chordal graphs. The spanner concept has a number of applications. For example, the sparse spanner of unweighted graphs is used in distributed computing and communication network design [2-5,13-14]. Cohen [7] once suggested a randomized parallel algorithm for finding a t-spanner with size $O(n^{1+\frac{2+\epsilon}{t}})$ on a weighted graph which needs $O(\frac{W_{max}}{W_{min}}\beta^2 \log^2 n)$ *expected time* with $O(n^{1/\beta} m\beta \log^2 n)$ work on an EREW PRAM, where $\beta = t/(2+\epsilon/2)$, where $wt(e)$ is the weight of edge e, $W_{max} = max\{wt(e) \mid e \in E\}$, $W_{min} = min\{wt(e) \mid e \in E\}$, and ϵ is a small constant.

Despite the existence of several efficient sequential and distributed algorithms for finding a sparse t-spanner of graphs, we have not seen any deterministic parallel algorithm for this problem. In this paper, we first relax the restriction of the problem by introducing an *approximate t-spanner* concept, and then present simple parallel algorithms for finding approximate t-spanners on both unweighted graphs and weighted graphs in terms of both size and weight. The algorithms exhibit a tradeoff between the running time and the factor of the spanner. Finally we present an NC algorithm for finding a $2t$-spanner on a weighted graph G.

The remaining parts of this paper are organized as follows. In Section 2 we introduce the tree decomposition concept. The algorithms for finding an approximate t-spanner for both unweighted graphs and weighted graphs with factors $O(t^{k+1})$ and $O(Dt^{k+1})$ respectively are presented in Section 3, where $D = max\{wt(e) \mid e \in MST\}$. If G is unweighted, the algorithm requires $O(\frac{n}{t^k} \log n)$ time and $M(n)$ processors, where $M(n)$ is the number of processors needed to find a Breadth-First Search tree in a graph with n vertices in time $O(\log n)$. The approximate t-spanner delivered has size of $O((\frac{n}{t^k})^{1+1/t} + n)$. Otherwise, the algorithm requires $O((\frac{n}{t^k})^2 + (\frac{n}{t^k})^{1+2/(t-1)} \log n)$ time and $O(n^2)$ processors. The approximate t-spanner delivered has size of $O((\frac{n}{t^k})^{1+2/(t-1)} + n)$, weight of $((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} + 1)wt(MST)$. In Section 4, we suggest an NC algorithm for constructing a $2t$-spanner on a weighted graph G with size $O(min\{m, \frac{n^2}{t} \log_{1+\epsilon}(\frac{W_{max}}{W_{min}})\})$ which requires $O(\log^3 n \log_{1+\epsilon}(\frac{W_{max}}{W_{min}}))$ time and $O(n^3)$ processors, where ϵ is a constant and $0 < \epsilon < 1/2$. All proposed parallel algorithms run on a CRCW PRAM in which simultaneous access by more than one processor to the same memory location for both read and write is allowed. In case several processors attempt to write in the same memory location simultaneously, an arbitrary one succeeds in doing the write.

2 Preliminaries

Let $\mathcal{X}_i \subseteq V$ and $\cup_{i=1}^{s} \mathcal{X}_i = V$, $1 \le i \le s \le n$. The set $\mathcal{H} = \{\mathcal{X}_i \mid 1 \le i \le s\}$ is called the *coarse vertex cover* on V if there exists $\mathcal{X}_i$ and $\mathcal{X}_j$ such that $\mathcal{X}_i \cap \mathcal{X}_j \neq \emptyset$, $i \neq j$. Otherwise $\mathcal{H}$ is called the *exact vertex cover* on V. If $\mathcal{H}$ is an exact vertex cover on V, then $\mathcal{X} \in \mathcal{H}$ is called a *vertex cluster*. Otherwise $\mathcal{X}$ is simply called a *partial cover*. For a vertex u, $\mathcal{X}$ is called u's *home cover* if $u \in \mathcal{X}$. Obviously, for an exact vertex cover $\mathcal{H}$, every vertex $u \in V$ has only one home cover. An *inverted tree* is a directed tree with the edges directed towards the root, and the root has a directed self-cycle.

Let $T(V, E_T)$ be an inverted tree with n vertices, we define the following restricted decomposition as *the tree decomposition* in which T is divided into a forest of n' inverted subtrees, and every inverted subtree has no more than *two* levels (the root of a tree is defined as the *first* level) where $n' \le \lfloor n/2 \rfloor$.

The tree decomposition can be easily implemented in parallel. We calculate the level number for every vertex in T. As a result, V is divided into two disjoint subsets V_1 consisting of all vertices with *odd* level numbers and V_2 consisting of all vertices with *even* level numbers. If $|V_1| \le |V_2|$ then all vertices in V_1 are selected as the roots of inverted subtrees. Let $u \in V_1$ be such a vertex, a vertex $v \in V_2$ belongs to the inverted subtree rooted at u *iff* the level number of v is larger than that of u by *one* and u is the parent of v in T. Otherwise the vertices in V_2 are selected as the roots of these inverted subtrees. For this case, we assign the root r of T to one of the inverted subtrees in which the root is one of r's children in T.

Lemma 2.1 Let $\mathcal{F}$ be a forest of inverted trees such that the number of vertices is n and each tree has two vertices at least. Then the tree decompositions in $\mathcal{F}$ can be finished in $O(\log n)$ time using $O(n)$ processors.

Proof. For every inverted tree T in $\mathcal{F}$, we first calculate its vertex level numbers and $|V_i|$, $i = 1, 2$. All of these operations can be done in $O(\log n)$ time with $O(n)$ processors. Then for every tree with $|V_1| \le |V_2|$, we apply the tree decomposition on it which can be done in $O(1)$ time with $O(n)$ processors. Finally we apply the tree decomposition to those trees with $|V_2| \le |V_1|$. Therefore the restricted tree decomposition in $\mathcal{F}$ can be finished in $O(\log n)$ time with $O(n)$ processors. □

3 Finding An Approximate t-spanner

3.1 Unweighted Graphs

3.1.1 Finding a $2t$-spanner

Awerbuch presented a distributed algorithm for constructing an optimal γ synchronizer in [2]. In the following we show that a $2t$-spanner with size $O(n^{1+1/t})$ can be achieved by the algorithm of Awerbuch. The basic idea of his algorithm is described as follows. The vertex set V is partitioned into maximum subsets of vertices called *clusters* such that every cluster is connected, and the diameter of every cluster does not exceed the *logarithm* of its cardinality. This guarantees that the total number of the neighboring cluster pairs is linear, and the maximum cluster diameter is logarithmic in the number of the vertices of the graph. The following is a parallel version of his algorithm.

Algorithm 1

```
1.  Initialization. $A(i) := 0$; $B(i) := 0$; $count := 0$.
/* $A(i) = 0$ means vertex $i$ is not in any cluster yet.*/
/* $B(i)$ means vertex $i$ is explored by cluster $B(i)$,*/
/* and count is the number of clusters. */
2. while there exists a vertex $i$ with $A(i) = 0$ do
2.1. $count := count + 1$; select a vertex $i$ such that
$A(i) = 0$ and $B(i) = max\{B(j) : j = 1, \ldots, n\}$;
$A(i) := 1$ and $B(i) := count$.
2.2. generate a BFS subtree $T_i$ rooted at $i$ such that
$T_i$'s diameter doesn't exceed the logarithm of its cardinality.
2.3. for every vertex $v$ in $T_i$, $A(v) := 1$.
for each $u$ in the rejected level of $T_i$, $B(u) := count$.
endwhile
```

Lemma 3.1. Let $G(V, E)$ be an unweighted graph. A $2t$-spanner of G with size $O(n^{1+1/t})$ can be generated in $O(n \log n)$ time with $M(n)$ processors.

Proof. By Algorithm 1, generating a cluster and labeling this cluster can be done in $O(\log n)$ time with $M(n)$ processors, currently the best result of $M(n) = n^{2.376}$ [9]. Selecting the center for a new cluster needs $O(\log n)$ time and $O(n/\log n)$ processors by prefix computation. While the number of iterations of the **while** loop is at most $O(n)$. Obviously the total number of edges in all BFS trees is at most $n-1$. By the step 2.2, for a cluster with n' vertices, there are at most pn' edges connected with other clusters. Therefore the resulting spanner has $O(n - 1 + pn) = O(n^{1+1/t})$ edges.

The factor of this spanner G' is considered as follows. For an edge $(u, v) \in E$ is not in G', if u and v are in the some cluster centered at w, $\mathcal{C}_w$, then the distance between u and v in G' is at most the distance between u and w plus the distance between v and w. So, the distance between u and v in G' is no more than $2\log_p n = 2t$. Otherwise $u \in \mathcal{C}_u$ and $v \in \mathcal{C}_v$, then the distance between u and v is one, and $(u, v) \in G'$ by the algorithm. □

3.1.2 Finding an approximate t-spanner

Our algorithm consists of several phases. The basic idea behind our algorithm is that, in each phase, we compress those vertices whose distances are not far away from each other into a *supervertex* (also called a cluster), form a *supergraph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ consists of all supervertices, and there is an edge in $\mathcal{E}$ *iff* there is at least one edge in G between two supervertices. So, the approximate spanner, denote by $SP(G)$, of G can be expressed as $SP(\mathcal{G}) \cup \{the\ tree\ edges\ in\ clusters\}$ recursively. In the following we give the detailed algorithm. The function D on $\mathcal{V}$ defines a forest of inverted trees in which $D(v)$ is the parent of v.

Algorithm 2

```
$i := 1$; $\mathcal{V} := V$.
while $i \leq k$ do
    for $j := 1$ to $\lfloor \log_3 t \rfloor$ do
    1. for each vertex $v$ in $\mathcal{V}$, set $D(v) := v$;
    2. for each vertex $v$, find a neighbor $u$ with
    the smallest index, $D(v) := u$;
    if $D(D(v)) = v$ and $D(v) \neq v$ then
      if $D(v) > v$ then $D(v) := v$ else $D(D(v)) := D(v)$.
    This leads to a forest $\mathcal{F}$ of inverted trees in
    which each tree has at least two vertices.
    3. generate another forest $\mathcal{F}'$ of inverted trees
    by applying the tree decomposition to $\mathcal{F}$.
    Denote by $E_{i,j}$ the edge set in $\mathcal{F}'$.
    4. construct a supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ such that
    each inverted tree in $\mathcal{F}'$ is a supervertex and
    an edge in $\mathcal{E}$ exists if there is an edge in $G$
    between vertices in these two supervertices.
    endfor
$i := i + 1$
endwhile
5. Find a $2t$-spanner $\mathcal{G}'$ of $\mathcal{G}(\mathcal{V}, \mathcal{E})$ by Algorithm 1.
6. The approximate $t$-spanner, $SP(G)$, of $G$ is
    $G(V, \cup_{i=1,\ldots,k}^{j=1,\ldots,\lfloor \log_3 t \rfloor} E_{i,j}) \cup \mathcal{G}'$.
```

Theorem 3.1. Given an unweighted, connected graph $G(V, E)$, an approximate t-spanner with factor $O(t^{k+1})$ can be constructed in $O(\frac{n}{t^k} \log n)$ time using $M(n)$ processors, and the spanner has size of $O((\frac{n}{t^k})^{1+1/t} + n)$, and factor of $O(t^{k+1})$, where k is a fixed constant, $1 \leq k \leq c \log_t n$ and $c < 1$.

Proof. By Algorithm 2, the **while** loop can be finished in $O(k \log_3 t \log n)$ time using at most $O(m + n)$ processors. The step 5 can be done in $O(\frac{n}{t^k} \log n)$ time using $M(n)$ processors by Lemma 3.1 which is also the dominant step of the entire algorithm.

By Lemma 2.1, the number of supervertices in a supergraph is at most *half* of the number of supervertices of its immediately precedent supergraph, assuming G is the initial supergraph. Let the number of supervertices of current supergraph be n_i ($n_1 = n$). Then the number of supervertices of the resulting supergraph after finishing the **for** loop is at most $O(\frac{n_i}{t})$. Therefore the final supergraph has $O(\frac{n}{t^k})$ supervertices after finishing the **while** loop. Following Lemma 3.1, the $2t$-spanner of the final supergraph has size $O((\frac{n}{t^k})^{1+1/t})$. Note that, for fixed i, $|E_{i,j}| \leq |E_{i,j-1}|/2$ and $|E_{1,1}| \leq n - 1$ because each $E_{i,j}$ is the edge set of a forest of inverted trees. So $\sum_{j=1}^{\lfloor \log_3 t \rfloor} |E_{i,j}| \leq O(\frac{n}{t^i})$. Therefore the size of the resulting approximate spanner is $O((\frac{n}{t^k})^{1+1/t} + n)$.

Now we calculate the factor. Let d_l be the maximum diameter of a cluster in the lth iteration of variable j for fixed i in the algorithm above. Initially every vertex in G is a supervertex and the diameter of every supervertex is d_0 $(= 0)$ when $i = 0$. The equation is described as follows.

$$d_l = 3d_{l-1} + 2, 1 \leq l \leq \lfloor \log_3 t \rfloor.$$

Then $d_{\lfloor \log_3 t \rfloor} = t - 1$ by the equation above. Therefore the maximum distance between two adjacent vertices in G belonging to the same supervertex is at most $O(t^k)$, and the factor of the resulting spanner is $O(2t(c't^k + 1)) = O(t^{k+1})$, where c' is a constant. □

3.2 Weighted Graphs

3.2.1 Finding a t-spanner

Assume that $d_G(x, y)$ is the weight of the shortest path between vertices x and y in graph G. Althőfer et al. [1] first considered the problem of finding a sparse t-spanner $G'(V, E')$ in an non-negative weighted graph $G(V, E)$, and presented a simple greedy algorithm for this problem described as follows.

Algorithm 3

```
1.  Initialization
    1.1. sort $E$ by non-decreasing weight;
    1.2. $G' := (V, \emptyset)$;
2. for every edge $e = (u, v)$ from the sorted list do
    2.1. compute $d_{G'}(u, v)$ between $u$ and $v$ in $G'$;
    2.2. if $d_{G'}(u, v) > t \times d_G(u, v)$
         then $E' := E' \cup \{(u, v)\}$; $G' := (V, E')$
    endfor
3. Output $G'$
```

If we use the fastest algorithm for finding the shortest path between two vertices [12], Algorithm 3 can be implemented in $O(mm + mn \log n) = O(n^4)$ time. The size of G' is $O(n^{1+2/(t-1)})$, and the weight is less than $(\frac{n}{t-1} + 1) wt(MST)$ [1]. By a considerably improved analysis of this algorithm, Chandra et al. [7] show that the running time of this algorithm is $O(n^{3+4/(t-1)})$, and the weight of G' is no more than $O(n^{\frac{2+\epsilon}{t-1}} wt(MST))$, where $\epsilon > 0$ is an any arbitrarily small constant.

The naive parallel version of Algorithm 3 requires $O(m + Rn \log n)$ time if $O(n^2)$ processors are available, where R is the size of G'. Here we observe that G' is such an augmented graph, each time we just put a new edge into it and re-calculate the shortest path between a pair of vertices on the augmented graph. Therefore, we can use the partially dynamic parallel algorithm for finding all pairs shortest paths, developed by Liang et al. [13], which claims that maintaining all pairs shortest paths can be done in $O(\log n)$ time using $O(n^2/ \log n)$ processors when inserting an edge to a graph. Hence we have

Lemma 3.2. Given a weighted, connected graph $G(V, E)$, finding a sparse t-spanner of G can be done in $O(m + n^{1+2/(t-1)} \log n)$ time using $O(n^2)$ processors. The size and the weight of the spanner generated are $O(n^{1+2/(t-1)})$ and $O(n^{\frac{2+\epsilon}{t-1}} wt(MST))$ respectively, where $\epsilon > 0$ is an any arbitrarily small constant.

Proof. Initially we compute the distance matrix of G. It can be easily done in $O(n \log n)$ time using $O(n^2)$ processors. Then we construct n single source shortest path trees for G' (see [13] for details). When a new edge is inserted into G', we update the distance matrix of G' which can be done in $O(1)$ time using $O(n^2)$ processors, then the maintenance of the data structures for n single source shortest path trees for G' requires $O(\log n)$ time and $O(n^2/\log n)$ processors. There are at most R edges to be inserted into G' and $R = O(n^{1+2/(t-1)})$ [1]. While deciding whether an edge of G belongs to G' can be done in $O(1)$ time by checking the corresponding entries in the distance matrices of both G and G'. Therefore this algorithm requires $O(m + n^{1+2/(t-1)} \log n)$ time provided that $O(n^2)$ processors are available. □

3.2.2 Finding an approximate t-spanner

The idea for finding an approximate t-spanner on weighted graphs is similar to that for unweighted graphs. The algorithm is same as the Algorithm 2 except the following steps.

Algorithm 4

```
        ⋮
2. for each vertex $v$, find a neighbor $u$
with the minimum weight, and set $D(v) := u$.
        ⋮
4. Construct a weighted supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$
such that each tree is a supervertex and $(u,v) \in E$
is selected as an edge of $\mathcal{E}$ if (i) $u \in \mathcal{C}_u$ and $v \in \mathcal{C}_v$;
(ii) the weight of the edge $(u,v)$ is minimum
among all such edges between $\mathcal{C}_u$ and $\mathcal{C}_v$;
        ⋮
5. Find a $t$-spanner $\mathcal{G}'$ on the resulting supergraph
$\mathcal{G}(\mathcal{V}, \mathcal{E})$ by Lemma 3.2.
        ⋮
```

Theorem 3.2. Given a weighted graph $G(V, E)$ with a fixed k, let D be the maximum edge weight of the MST of G. Then finding an approximate t-spanner with factor $O(Dt^{k+1})$ requires $O(\frac{n}{t^k})^2 + (\frac{n}{t^k})^{1+2/(t-1)} \log n)$ time and $O(n^2)$ processors. The approximate t-spanner generated has size of $O((\frac{n}{t^k})^{1+2/(t-1)} + n)$, weight of $O(((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} + 1) wt(MST))$, where $\epsilon > 0$ is an any arbitrarily small constant.
Proof. Similar to Theorem 3.1, omitted. □

4 NC Algorithms for 2t-spanners

4.1 Unweighted graphs

In the following we present an NC algorithm for finding a $2t$-spanner of unweighted graphs G with size $O(\max\{m, n^2/t\})$. The motivation that we develop this algorithm is to use it as a subroutine to devise an NC algorithm for finding a $2t$-spanner on weighted graphs.

Let $G(V, E)$ be an unweighted graph and $d(u, v)$ the distance between vertices u and v in G. Assuming that G does not contain degree-one vertices. Otherwise we can delete these vertices first. We then put them back to the resulting t-spanner of the graph formed by the remaining vertices. We construct a new graph $G_i(V, E_i)$ from the graph $G(V, E)$ such that an edge $(u, v) \in E_i$ if and only if $d(u, v) \le 2^i$, $0 \le i \le \log n$.

Denote by U_i a maximal independent set of vertices on $G(V, E_i)$, and $T(u, i)$ a Breadth-First Search tree rooted at u with height 2^i in $G(V, E)$.
Lemma 4.1. Given the graph $G(V, E)$, the graph $G_i(V, E_i)$ can be constructed in $O(i)$ time using $O(n^3)$ processors, $0 \le i \le \log n$.
Lemma 4.2. For any two non-adjacent vertices u and v in the graph $G_{\lfloor \log t \rfloor -1}(V, E_{\lfloor \log t \rfloor -1})$, $d(u, v) > 2^{\lfloor \log t \rfloor -1} \ge t/4$.
Lemma 4.3. $\frac{n}{\Delta^{\lfloor \log t \rfloor -1}+1} \le |U_{\lfloor \log t \rfloor -1}| \le \frac{4n}{t}$.

Algorithm 5

```
1. Construct the graph $G_{\lfloor \log t \rfloor -1}(V, E_{\lfloor \log t \rfloor -1})$.
2. Find a $U_{\lfloor \log t \rfloor -1}$ in $G_{\lfloor \log t \rfloor -1}$.
3. For every $u \in U_{\lfloor \log t \rfloor -1}$, construct the $T(u, \lfloor \log t \rfloor)$.
4. The $2t$-spanner of $G$ is $\cup_{u \in U_{\lfloor \log t \rfloor -1}} T(u, \lfloor \log t \rfloor)$.
```

Let $Cover(u, \lfloor \log t \rfloor)$ be a set consisting of all vertices in $T(u, \lfloor \log t \rfloor)$, and u be defined as the *center* of this set. By the definition in Section 2, $Cover(u, \lfloor \log t \rfloor)$ is a partial cover and $\mathcal{H} = \{Cover(u, \lfloor \log t \rfloor) \mid u \in U_{\lfloor \log t \rfloor -1}\}$ is a coarse vertex cover on V.
Theorem 4.1. The spanner generated by Algorithm 5 has size of $O(min\{m, n^2/t\})$, and factor of $2t$.
Proof. By Lemma 4.3, $|U_{\lfloor \log t \rfloor -1}| \le 4n/t$, and a vertex $v \in V$ belongs to at most $4n/t$ home covers, and therefore the size of the resulting spanner is $O(min\{m, \frac{n^2}{t}\})$.

Now we show that the factor of the resulting spanner is $2t$. To achieve this bound, we only show that for every two vertices u and v such that $d(u, v) \le 2^{\lfloor \log t \rfloor -1}$, there exists at least a home cover $\mathcal{X} \in \mathcal{H}$ including these two vertices. That means, the distance between u and v in the spanner is no more than $2t$. The proof is as follows. By Lemma 4.2, it is impossible that both u and v are in $U_{\lfloor \log t \rfloor -1}$. So we discuss it by four cases. (i) If either one of them is in $U_{\lfloor \log t \rfloor -1}$, say $u \in U_{\lfloor \log t \rfloor -1}$, then v is in the home cover of u because $d(u, v) \le 2^{\lfloor \log t \rfloor}$. As a result, the distance between these two vertices in the spanner is no more than $2^{\lfloor \log t \rfloor} \le t$. (ii) If neither u nor v is in $U_{\lfloor \log t \rfloor -1}$ but their common neighbor w of $G_{\lfloor \log t \rfloor -1}$ is in $U_{\lfloor \log t \rfloor -1}$, then w is the center of their home cover because $d(v, w) \le 2^{\lfloor \log t \rfloor -1}$ and $d(u, w) \le 2^{\lfloor \log t \rfloor -1}$, the theorem follows. (iii) If w is a neighboring vertex of u but v in $G_{\lfloor \log t \rfloor -1}$. We claim that v also belongs to the home cover centered at w, because $d(v, w) \le d(w, u) + d(u, v) \le 2^{\lfloor \log t \rfloor -1} + 2^{\lfloor \log t \rfloor -1} \le 2^{\lfloor \log t \rfloor}$. (iv) Otherwise, the distance between a vertex in $U_{\lfloor \log t \rfloor -1}$ and u (or v) in $G_{\lfloor \log t \rfloor -1}$ is at least larger than 2. Then either u or v must belong to $U_{\lfloor \log t \rfloor -1}$ because $U_{\lfloor \log t \rfloor -1}$ is a maximal independent set of graph $G_{\lfloor \log t \rfloor -1}$. Contradiction. □.

4.2 Weighted graphs

Cohen [7] introduced the *pairwise cover* concept, and presented an efficient sequential algorithm and a randomized parallel algorithm for finding a sparse t-spanner in weighted graphs. The basic idea of Cohen's algorithm is to employ a logarithmic number of pairwise covers for different values of W to construct spanners. Define the *radii* of a partial cover $\mathcal{X}$ by the distance from the center of $\mathcal{X}$ to the farthest vertex in it. Let ϵ' be any constant bounded by $0 < \epsilon' < 1/2$. Cohen's algorithm is described as follows.

Algorithm 6

```
1.  Initialization.
    1.1. $W_{max} := max\{wt(e) \mid e \in E\}$;
         $W_{min} := min\{wt(e) \mid e \in E\}$;
         $R := \lceil \log_{1+\epsilon'}(\frac{W_{max}}{W_{min}}) \rceil$;
    1.2. for $i := 0$ to $R$ do in parallel
         $W_i := W_{min}(1+\epsilon')^i$
         endfor
2.  for $i := 0$ to $R$ do in parallel
    Construct a coarse vertex cover $\mathcal{H}_i$ such that
    the radii of every partial cover $\mathcal{X} \in \mathcal{H}_i$ is
    no more than $2^{\lfloor \log t \rfloor} W_i$.
    endfor
3.  The spanner generated is
$\cup_{i=1}^{r} \{T_i(u, \lfloor \log t \rfloor) \mid u\ is\ the\ center\ of \mathcal{X}\ and\ \mathcal{X} \in \mathcal{H}_i\}$.
```

The key part of this algorithm is how to efficiently construct a coarse vertex cover on V (called pairwise cover in Cohen's algorithm) in parallel. As for this, Cohen [8] introduces randomness to chose the centers of covers. However, it seems not easy to transform this randomized parallel algorithm into a deterministic version. Here we present an efficient, deterministic parallel algorithm for constructing such a coarse vertex cover by extending our technique for unweighted graphs to weighted graphs. Let A be the adjacent weighted matrix of G and $A^i = A^{i-1} \bigodot A^{i-1}$, where $\bigodot$ operation is defined as follows: $a^i_{(u,v)} = min_{w \in V}\{a^{i-1}_{(u,v)},\ a^{i-1}_{(u,w)} + a^{i-1}_{(w,v)}\}$ and an entry $a^i_{(u,v)}$ of A^i represents the distance between u and v with at most 2^i edges. Let $A^0 = A$. Then a coarse vertex cover $\mathcal{H}_i$ is constructed as follows.

Algorithm 7

```
1. Construct an auxiliary graph $G_i(V, E^*)$, an edge
    $(u, v)$ is added to $E^*$ iff $d(u, v) \le 2^{\lfloor \log t \rfloor - 1} W_i$.
2. Find a maximal independent set $U(i, \lfloor \log t \rfloor - 1)$ of
    $G_i(V, E^*)$ in parallel.
3. Build a shortest path tree rooted at $u$, $T_i(u, \lfloor \log t \rfloor)$
    with height $2^{\lfloor \log t \rfloor} W_i$ in $G(V, E)$ for each
    $u \in U(i, \lfloor \log t \rfloor - 1)$ such that a vertex $v$ is
    included in this tree iff $d(u, v) \le 2^{\lfloor \log t \rfloor} W_i$.
4. The coarse vertex cover is built, where
$\mathcal{H}_i = \{Cover_i(u, \lfloor \log t \rfloor) \mid u \in U(i, \lfloor \log t \rfloor - 1)\}$.
```

Lemma 4.7. Given a weighted graph $G(V, E)$ with respect to parameters t and W_i, it can be done in $O(\log^3 n)$ time using $O(n^3)$ processors for constructing a coarse vertex cover $\mathcal{H}_i$ such that two vertices u and u are included in one home cover $\mathcal{X}$ at least if $d(u, v) \le 2^{\lfloor \log t \rfloor - 1} W_i$ for fixed i, where $\mathcal{X} \in \mathcal{H}_i$.

Proof. By Algorithm 7, the step 1 can be finished in $O(\log n \log t)$ time using $O(n^3)$ processors. The details are as follows. First compute the matrix $A^{\lfloor \log t \rfloor - 1}$ which requires $O(\log n \log t)$ time using $O(n^3)$ processors. Then construct the graph $G_i(V, E^*)$ which requires $O(1)$ time using $O(n^3)$ processors. The step 2 can be done in $O(\log^3 n)$ time using $O(n^2/\log n)$ processors by the algorithm of Goldberg and Spencer [18]. The processing of the step 3 is more involved. For every vertex $u \in U(i, \lfloor \log t \rfloor - 1)$, we build an inverted tree rooted at u, $T_i(u, \lfloor \log t \rfloor)$, initially. Then we check a vertex $v \in V$ to see whether $v \in T_i(u, \lfloor \log t \rfloor)$ by testing $a^i_{(u,v)} \le 2^{\lfloor \log t \rfloor} W_i$. If it does, we find the parent $p(u, v)$ of v in this tree, where $p(u, v)$ is such a vertex w that $a^i_{(u,v)} = a^i_{(u,w)} + wt((w, v))$. There are at most $|U(i, \lfloor \log t \rfloor - 1)| \le n$ trees, and for each vertex finding its parent in a tree can be done in $O(\log n)$ time using $O(n)$ processors. So, the step 3 requires $O(\log n)$ time and $O(n^2)$ processors. Step 4 can be done in $O(1)$ time using $O(n^2)$ processors. □.

Theorem 4.2. For a weighted connected graph $G(V, E)$ with non-negative weights, the spanner generated by Algorithms 6 and 7 has size of $O(min\{m, \frac{n^2}{t} \log_{1+\epsilon}(\frac{W_{max}}{W_{min}})\})$ and factor of $2t$, where ϵ is a constant and $0 < \epsilon < 1/2$.

Proof. Similar to Theorem 4.1, omitted. □

References

[1] I. Althőfer, G. Das, D. Dobkin, D. Joseph and J. Soares, On sparse spanners of weighted graphs, *Disc. and Comp. Geometry*, Vol. 9, 1993, 81-100.

[2] B. Awerbuch, Complexity of network synchronization, *J. ACM*, Vol. 32, 1985, 805-823.

[3] B. Awerbuch, A. Bar-Noy, N. Linial and D. Peleg, Compact distributed data structures for adaptive routing, *Proc. 21st ACM Sympo. on Theory of Computing*, 1989, 479-489.

[4] B. Awerbuch and D. Peleg, Routing with polynomial communication-space trade-off, *SIAM J. Discrete Math.*, Vol. 5, 1992, 151-162.

[5] B. Awerbuch and D. Peleg, Sparse partitions, *Proc. 31st IEEE Sympo. on Founda. of Computer Sci.*, 1990, 501-513.

[6] B. Chandra, G. Das, G. Narasimhan and J. Soares, New sparseness results on graph spanners, *Proc. 8th ACM Sympo. on Comput. Geometry*, 1992, 192-201.

[7] E. Cohen, Fast algorithms for constructing t-spanners and paths with t, *Proc. 34th IEEE Sympo. on Founda. of Comput. Sci.*, 1993, 648-658.

[8] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *Proc. 19th ACM Sympo. on Theory Comput.*, 1987, 1-6.

[9] G. Das and G. Narasimhan, A fast algorithm for constructing sparse Euclidean spanners, *Proc. 10th ACM Sympo. on Comput. Geometry*, 1994, 132-139.

[10] M. L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimalization problems, *J. ACM*, Vol.34, 1987, 596-615.

[11] W. Liang, B. McKay and H. Shen, NC algorithms for dynamically solving the all pairs shortest path problem and related problems, Unpublished manuscript, Aug., 1995.

[12] D. Peleg and A. A. Schăffer, Graph spanners, *J. Graph Theory*, Vol. 13, 1989, 99-116.

[13] D. Peleg and J. Ullman, An optimal synchronizer for the hypercube, *SIAM J. Comput.*, Vol. 18, 1989, 740-747.

[14] D. Peleg and E. Upfal, A trade-off between space and efficiency for routing tables, *J. ACM*, Vol. 36, 1989, 510-530.

[15] M. Goldberg and T. Spencer, Constructing a maximal independent set in parallel, *SIAM J. Discrete Math.*, Vol. 2, 1989, 322-328.