THE AUSTRALIAN NATIONAL UNIVERSITY

**TR-CS-96-01**

# Constructing the Spanners of Graphs in Parallel

**Weifa Liang and Richard P. Brent**

**January 1996**

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

**Recent reports in this series:**

TR-CS-95-08 David Hawking. *The design and implementation of a parallel document retrieval engine*. December 1995.

TR-CS-95-07 Raymond H. Chan and Michael K. Ng. *Conjugate gradient methods for Toeplitz systems*. September 1995.

TR-CS-95-06 Oscar Bosman and Heinz W. Schmidt. *Object test coverage using finite state machines*. September 1995.

TR-CS-95-05 Jeffrey X. Yu, Kian-Lee Tan, and Xun Qu. *On balancing workload in a highly mobile environment*. August 1995.

TR-CS-95-04 Department of Computer Science. *Annual report 1994*. August 1995.

TR-CS-95-03 Douglas R. Sweet and Richard P. Brent. *Error analysis of a partial pivoting method for structured matrices*. June 1995.

# Constructing the Spanners of Graphs in Parallel[1]

Weifa Liang† Richard P. Brent‡

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

‡ Computer Sciences Lab.
The Australian National University
Canberra, ACT 0200, Australia
rpb@discus.anu.edu.au

**Abstract**

Given a connected graph $G = (V, E)$ with $n$ vertices and $m$ edges, the distance between two vertices in $G$ is the weight of the shortest path between them. 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 variable $t$ and $t \leq f(t) < n$. In this paper parallel algorithms for finding approximate $t$-spanners on both unweighted graphs and weighted graphs are given. If $G$ is an unweighted graph, our algorithm requires $O(\frac{n}{t^k} \log n)$ time and $M(n)$ processors, and the spanner generated has size of $O((\frac{n}{t^k})^{1+1/t} + n)$ and factor of $O(t^{k+1})$; otherwise our algorithm requires $O((\frac{n}{t^k})^2 + (\frac{n}{t^k})^{1+2/(t-2)} \log n)$ time and $O(n^2)$ processors, and the spanner generated has size of $O((\frac{n}{t^k})^{1+2/(t-1)} + n)$, weight of $((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} + 1)wt(MST)$, and factor of $O(Dt^{k+1})$, where $M(n)$ is the number of processors needed to find a Breadth-First Search tree in a graph with $n$ vertices in time $O(\log n)$, $D$ is the maximum edge weight of a minimum spanning tree of $G$, and $k$ is a fixed constant integer with $1 \leq k \leq \log_t n$. Also, an NC algorithm for finding a $2t$-spanner on a weighted graph $G$ with size $O(min\{m, \frac{n^2}{t} \log_{1+\epsilon}(\frac{W_{max}}{W_{min}})\})$ is presented. The proposed algorithm requires $O(\log^3 n \log_{1+\epsilon}(\frac{W_{max}}{W_{min}}))$ time and $O(n^3)$ processors, where $W_{max} = max\{wt(e) \mid e \in E\}$, $W_{min} = min\{wt(e) \mid e \in E\}$, and $\epsilon$ is a constant with $0 < \epsilon < 1/2$. All algorithms are for a CRCW PRAM model.

**Keywords:** Approximate spanners, parallel algorithms, graph problems.

[1] A short version will appear in *Proc. of 10th Intern. Conf. on Parallel Processing Sympo.*, 1996

# 1 Introduction

Given a connected graph $G = (V, E)$ with $n$ vertices and $m$ edges, 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 variable $t$ 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.

Regarding the spanners on some special graphs such as Euclidean graphs, geometry graphs and chordal graphs, much effort has been taken in recent several works [2,7,10,11,14,17]. Peleg and Schäffer [14], Althöfer, Das, Dobkin, Joseph and Soares [1] have shown some very interesting characters about the $t$-spanner of both unweighted graphs and weighted graphs. Besides, 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 [3,4,5,6,15,16]. Peleg and Upfal [16], Awerbuch Bar-Noy, Linial and Peleg [4], and Awerbuch and Peleg [5] use it to design efficient routing schemes in distributed networks. Peleg and Ullman [15] also point out that the $t$-spanner is useful to design a synchronizer [3]. Whereas Cohen [8] suggests 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 $\epsilon$ is an arbitrarily small constant.

Despite existing 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 *approximate t-spanner* concept, and then present simple parallel algorithms for finding a sparse, approximate $t$-spanner on both unweighted graphs and weighted graphs in terms of both size and weight. The algorithms exhibit a trade-off 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 some notations and 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 $O(Dt^{k+1})$ are presented in Section 3, where $D = max\{wt(e)|e \in MST\}$. If $G$ is an unweighted graph, 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 by this algorithm 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 by this algorithm 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 $O(min\{m, \frac{n^2}{t}\log_{1+\epsilon}(\frac{W_{max}}{W_{min}})\})$. This algorithm requires $O(\log^3 n \log_{1+\epsilon}(\frac{W_{max}}{W_{min}}))$ time and $O(n^3)$ processors, where $\epsilon$ is a small constant with $0 < \epsilon < 1/2$. We conclude in Section 5 by leaving an open problem.

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

## 2.1 Notations

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-loop.

## 2.2 Tree decomposition

In order to make our algorithmic description easier in section 3, we first introduce a restricted tree decomposition concept. Let $T(V, E_T)$ be an inverted tree with $n$ vertices, we define the following restricted decomposition as *the tree decomposition* in which each inverted tree $T$ is divided into a forest of $n'$ inverted subtrees, and every 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 defined above can be easily implemented in parallel. We calculate the level number for every vertex in $T$. After that, the vertex set $V$ 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 subtrees. Let $u \in V_1$ be such a vertex, a vertex $v \in V_2$ belongs to the subtree rooted at $u$ if and only if the level number of $v$ is larger than that of $u$ by *one* and $u$ is the parent of $v$ in $T$. Otherwise we select the vertices in $V_2$ as the roots of these subtrees. For this case, we assign the root $r$ of $T$ to one of the subtrees in which the root is one of $r$'s children in $T$.

**Lemma 1** *Let $\mathcal{F}$ be a forest in which the number of vertices is $n$ and each tree has two vertices at least. Then implementing all tree decompositions in $\mathcal{F}$ can be finished in $O(\log n)$ time using $O(n)$ processors.*

**Proof** For every 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 $|V_1| \leq |V_2|$, 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 $|V_2| \leq |V_1|$. Therefore the restricted tree decomposition in $\mathcal{F}$ can be finished in $O(\log n)$ time with $O(n)$ processors.□

# 3 Parallel Algorithms for Finding Approximate t-Spanners

## 3.1 Unweighted Graphs

### 3.1.1 The algorithm for finding a $2t$-spanner

Peleg and Schäffer [14] once gave the following lemma, and proved that the size of the spanner is tight in big-O sense.

**Lemma 2** *[14]. For every $n$-vertex graph $G$ and for every fixed $t \geq 1$, there exists a polynomial time constructible $(4t + 1)$-spanner with $O(n^{1+1/t})$ edges.*

Actually Lemma 2 is derived directly from Awerbuch's algorithm of constructing an optimal $\gamma$ synchronizer [3]. 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 graph's vertices.

In the following, the *remaining graph* denotes the subgraph induced by the vertices that were not yet joined to clusters. The basic phase of his algorithm is that a vertex in the remaining graph is chosen as the new cluster center, and a cluster centered at this vertex is formed. This phase is repeated until there are no more vertices in the remaining graph. In details, let $p = n^{1/t}$, first select an arbitrary vertex $v$ as the root of a Breath-First Search (BFS) tree, and then generate this BFS tree level by level until it reaches some level $l_0$ such that the number of vertices in this level is less than $p - 1$ times the total number of vertices in all previous levels, and the level $l_0$ is said the *rejected level*. At that moment, the procedure for generating clusters is terminated, and all vertices with level number $l \leq l_0$ belong to the cluster $\mathcal{C}_v$ centered at $v$. After a cluster $\mathcal{C}$ formed, its rejected level is examined. If it is not empty, then a vertex $u$ in this rejected level is chosen as the new cluster center. In case the rejected level of cluster $\mathcal{C}$ is empty, the algorithm backtracks to the cluster

from which $\mathcal{C}$ itself is found. The resulting spanner $SP(G) = (V, E_1 \cup E_2)$ of $G$ is constructed as follows:

$$E_1 = \cup_{all\ clusters\ \mathcal{C}} \{e : e \text{ is an edge in the } BFS \text{ tree of a cluster } \mathcal{C}\}$$
$$\text{and } E_2 = \{(u, v) : (u, v) \in E,\ u \in \mathcal{C}_u \text{ and } v \in \mathcal{C}_v\}.$$

So the size of this spanner is $|E_1| + |E_2| \leq (n-1) + pn = O(n^{1+1/t})$. We leave the analysis of the factor of this spanner later.

The parallelization of Awerbuch's algorithm is not very difficult. Here we give a parallel version of his algorithm.

**Algorithm 1**

Step 1. Initialization.
$A(i) := 0$; $B(i) := 0$; $count := 0$.
/* $A(i) = 0$ means vertex $i$ does not belong to any clusters yet.*/
/* $B(i)$ means vertex $i$ is explored by the $B(i)$th cluster */
/* $count$ is the number of clusters in $G$ */

Step 2. **while** there exists a vertex $i$ with $A(i) = 0$ **do**
Step 2.1. $count := count + 1$; select a vertex $i$ with $A(i) = 0$ and $B(i) = max\{B(j) : j = 1, \ldots, n\}$ as the root of of the new BFS tree, set $A(i) := 1$ and $B(i) := count$.
Step 2.2. Generate a BFS subtree $T_i$ rooted at $i$ such that the diameter of $T_i$ doesn't exceed the logarithm of its cardinality.
Step 2.3. for every vertex $v$ in $T_i$, set $A(v) := 1$.
Step 2.4. for every vertex $v$ in the rejected level of $T_i$, set $B(v) := count$.
**endwhile**

Now we have the following lemma.

**Lemma 3** *Let $G(V, E)$ be an unweighted, connected graph. Then 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, the initialization can be done in $O(1)$ time using $O(n)$ processors. Then generating a cluster and labeling this cluster can be done in $O(\log n)$ time with $M(n)$ processors. Note that $M(n) \geq \Omega(n^2)$, currently the best result of $M(n) = n^{2.376}$ [9]. Finally selecting the center for the 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 definition, for a cluster with $n'$ vertices, there are at most $pn'$ edges connected with other clusters. Therefore the resulting spanner has $O(pn) = O(n^{1+1/t})$ edges.

The factor of this spanner $G'$ is considered as follows. For an edge $(u, v) \in E$ not in $G'$, if $u$ and $v$ are in the some cluster centered at $w$, $\mathcal{C}_w$, then the distance between $u$ and $v$ is at most the distance between $u$ and $w$ plus the distance between $v$ and $w$. So, the distance between $u$ and $v$ is no more than $2 \log_p n = 2t$. In case $u \in \mathcal{C}_u$ 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 The algorithm for 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 $\mathcal{E}$ if and only if there is at least one edge in $G$ between two supervertices (we keep track of its corresponding edge in $G$). 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**
    Step 1. for each vertex $v$ in $\mathcal{V}$, set $D(v) := v$;
    Step 2. for each vertex $v$, find an neighbor $u$ with the smallest index,
      and set $D(v) := u$ ;
    if $D(D(v)) = v$ and $D(v) \neq v$ then
      if $D(v) > v$ then $D(v) := v$ else $D(D(v)) := D(v)$.
    As a result, we obtain a forest $\mathcal{F}$ of inverted trees.
    /* Now there is a forest $\mathcal{F}$ containing all inverted trees formed in Step 2, */
    /* and each inverted tree has at least *two* vertices */
    Step 3. generate another forest $\mathcal{F}'$ by applying the tree decomposition to $\mathcal{F}$.
      Denote by $E_{i,j}$ the edge set in $\mathcal{F}'$.
      /* Now every tree is partitioned into many inverted subtrees */
    Step 4. Construct a supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ such that each inverted subtree in $\mathcal{F}'$
      is a supervertex and there is an edge in $\mathcal{E}$ if there is an edge
      between two vertices in these two supervertices.
  **endfor**
  $i := i + 1$
**endwhile**
Step 5. Find a $2t$-spanner $\mathcal{G}'$ of the resulting supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in parallel
    by applying the parallel algorithm in Section 3.1.
Step 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 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.*

**Proof** By the proposed algorithm above, we know that the Step 1–Step 4 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, and this step is also the dominant step of the proposed algorithm. □

**Theorem 2** *The approximate t-spanner generated by Algorithm 2 has size of* $O((\frac{n}{t^k})^{1+1/t}+n)$, *and the 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 Lemma 1, we know that the number of supervertices in a supergraph is at most *half* of the number of vertices of its immediately precedent supergraph, assuming $G$ is an 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 3.2, the $2t$-spanner of the final supergraph has size $O((\frac{n}{t^k})^{1+1/t})$. We also 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 $l$th 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.

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

From the equation above we derive that $d_{\lfloor \log_3 t \rfloor} = t - 1$. 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. □

**Corollary 1** *Given a connected unweighted graph* $G(V, E)$, *a* $(4t+1)$*-spanner with size* $O(min\{m, (\frac{n}{t})^2\})$ *can be generated in* $O(\log n)$ *time using* $O(m+n)$ *processors.*

**Proof** By setting $k = 1$, run the Step 1–Step 4 of Algorithm 2, we get a supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Having this supergraph, a $(4t+1)$-spanner $G(V, E_1 \cup E_2)$ of $G$ is constructed as follows. $E_1 = \cup_{j=1}^{\lfloor \log_3 t \rfloor} E_{1,j}$, and only one edge $(x_0, y_0)$ is selected into $E_2$ for all edges $(x, y) \in E$, $x \in \mathbf{v_1}$, $y \in \mathbf{v_2}$, and $\mathbf{v_i} \in \mathcal{V}$ for $i = 1, 2$. So, the size of the resulting spanner is $|E_1| + |E_2| \leq 2n + (c_1 \frac{n}{t})^2 = O(min\{m, (\frac{n}{t})^2\})$. The factor of this spanner is analyzed as follows. Consider an edge $(x, y) \in E$ but $(x, y) \notin E_1 \cup E_2$. If $x,\ y \in \mathbf{v} \in \mathcal{V}$, then the distance between them in the spanner is at most $2t$. Otherwise, let $x,\ x_0 \in \mathbf{v_1}$ and $y,\ y_0 \in \mathbf{v_2}$ and the edge between $\mathbf{v_1}$ and $\mathbf{v_2}$ be $(x_0, y_0)$, then the distance between $x$ and $y$ in the spanner is the summation of the distance between $x$ and $x_0$, plus the distance between $y$ and $y_0$, and plus one (the edge $(x_0, y_0)$) which equals at most $4t + 1$. □

## 3.2 Weighted Graphs

### 3.2.1 The algorithm for finding a $t$-spanner

Althőfer, Das, Dobkin, Joseph and Soares [2] first consider the problem of finding a sparse $t$-spanner $G'(V, E')$ in an non-negative weighted graph $G(V, E)$, and present a simple greedy algorithm for this problem described as follows. Let $d_G(x, y)$ be the weight of the shortest path between vertices $x$ and $y$ in graph $G$.

**Algorithm 3**

Step 1. Initialization
  Step 1.1. Sort $E$ by non-decreasing weight;
  Step 1.2. $G' := (V, \emptyset)$;
Step 2. **for** every edge $e = (u, v)$ in $E$ (from the sorted list) **do**
  Step 2.1. Compute $P(u, v)$, the shortest path between $u$ and $v$ in $G'$;
  Step 2.2. **if** $d_{G'}(u, v) > t \times d_G(u, v)$ **then** $E' := E' \cup \{(u, v)\}$;
    $G' := (V, E')$
  **endfor**
Step 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, Das, Narasimhan and Soares [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 $G'$. Therefore we obtain a simple parallel algorithm which needs $O(m + Rn \log n)$ time with $O(n^2)$ processors because finding a shortest path between two vertices requires $O(n \log n)$ time using $O(n^2)$ processors. But 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 this augmented graph. Therefore, we can use the partially dynamic parallel algorithm for finding all pairs shortest paths, developed by Liang, McKay and Shen [13], which claims that maintaining all pairs shortest paths can be done in $O(\log n)$ time using $O(n^2 / \log n)$ processors. Hence we obtain the following lemma.

**Lemma 4** *Given a weighted, connected graph $G(V, E)$, finding a sparse $t$-spanner of $G$ can be done in $O(m + R \log n) = O(m + n^{1+2/(t-1)} \log n)$ 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 maintaining 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 time $O(m + n^{1+2/(t-1)} \log n)$ time provided that $O(n^2)$ processors are available.□

### 3.2.2 The algorithm for finding an approximate $t$-spanner

The idea for finding an approximate $t$-spanner on weighted graphs is similar to that for unweighted graphs. However, we must make some minor changes in order to extend our algorithm for unweighted graphs to weighted graphs. At this moment, we concentrate on the construction of the supergraph in each phase. Instead of finding an arbitrary edge between two supervertices served as an edge in the resulting supergraph, an edge of $G$ between two supervertices is selected as the edge in the supergraph when it has the minimum weight among all edges in these two supervertices. The algorithm is the same as Algorithm 2 except the following steps.

**Algorithm 4**

$\vdots$

Step 2. for each vertex $v$, find such an adjacent neighbor $u$ with the minimum weight compared with other neighbors, and set $D(v) := u$.
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).
As a result, we obtain a forest $\mathcal{F}$ of inverted trees.

$\vdots$

Step 4. Construct a weighted supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ such that each subtree is a supervertex and the edge $(u, v)$ 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 $\mathcal{C}_v$;

$\vdots$

Step 5. Find a $t$-spanner $\mathcal{G}'$ on the resulting weighted supergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$

$\vdots$

Now we analyze the computational complexity of the proposed algorithm.

**Theorem 3** *Given a connected, weighted graph $G(V, E)$, let $D$ be the maximum edge weight of the minimum spanning tree of $G$. Then finding an approximate, sparse $t$-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.*

**Proof** By Algorithm 4, we know that the Step 1–Step 4 can be finished in $O(k \log_3 t \log n)$ time using $O(m+n)$ processors. By Lemma 4, the Step 5 can be done in $O((\frac{n}{t^k})^2 + (\frac{n}{t^k})^{1+2/(t-1)} \log n)$ time using $O(n^2)$ processors. □

**Theorem 4** *Let $SP(G)$ be the approximate $t$-spanner of $G$ generated by Algorithm 4 for fixed $k$. The size of $SP(G)$ is $O((\frac{n}{t^k})^{1+2/(t-1)} + n)$, the weight is no more than $O(((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} + 1)wt(MST))$ and the factor is $O(Dt^{k+1})$, where $\epsilon > 0$ is an any arbitrarily small constant.*

**Proof** By the same argument in Section 3.1.2, we know that the number of supervertices in a weighted supergraph is at most *half* of the number of that in its immediately precedent supergraph, assuming $G$ is the initial supergraph. Let the current supergraph have $n_i$ vertices ($n_1 = n$). Then the number of supervertices of the resulting supergraph after finishing the **for** loop is at most $O(\frac{n_i}{t})$. We therefore obtain the final supergraph with $O(\frac{n}{t^k})$ supervertices after finishing the **while** loop. 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. So $\sum_{j=1}^{\lfloor \log_3 t \rfloor} |E_{i,j}| \leq O(\frac{n}{t^i})$. Note that that $\cup_{i=1,\ldots,k}^{j=1,\ldots,\lfloor \log_3 t \rfloor} E_{i,j} \subset \{e : e \text{ is an edge in } MST\}$. Therefore after finishing Step 4, we obtain a supergraph, denote by $G_1$, with $O(\frac{n}{t^k})$ vertices. By applying the analysis in [1,7]. we know the weight of the $t$-spanner of $G_1$ is less than $O((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} wt(MST_1)) = O((\frac{n}{t^k})^{\frac{2+\epsilon}{t-1}} wt(MST))$ because $wt(MST_1) \leq wt(MST)$, and the size of $G_1$ is $O((\frac{n}{t^k})^{1+2/(t-2)})$ where $MST_1$ is the minimum spanning tree of $G_1$.

Now we calculate the factor of the spanner. Let $d_l$ be the maximum diameter of the cluster generated in the $l$th iteration of the variable $j$ for fixed $i$. Initially the diameter of every supervertex is at most $d_1$ when $i = 1$, where $0 \leq d_1 \leq D$. The equation is expressed as follows.

$$d_i = 3d_{i-1} + 2,\ 2 \leq i \leq \lfloor \log_3 t \rfloor$$

From the equation above we derive that $d_{\lfloor \log_3 t \rfloor} = \frac{t}{3}(D+1) - 1 = O(Dt)$. Therefore the maximum distance of two adjacent vertices of $G$ belonging to the same supervertex in $G_1$ is at most $O(Dt^k)$. Thus, the factor of the resulting spanner of $G$ is $O(t(Dt^k + 1)) = O(Dt^{k+1})$. □

# 4 NC Algorithms for Finding A $2t$-spanner of $G$

## 4.1 The algorithm for unweighted graphs

In Corollary 1 we already obtained an NC algorithm for finding a $(4t+1)$-spanner of unweighted graphs with size $O((\frac{n}{t})^2)$. By adjusting the upper bound of $j$ in Algorithm 2 to $\lfloor \log_3 t \rfloor / 2$, we can easily derive an NC algorithm for finding a $(2t+1)$-spanner with size $O((\frac{n}{t})^2)$. In the following we present an NC algorithm for finding

a $2t$-spanner of $G$ with size $O(\max\{m, n^2/t\})$ which is obviously worser than the result in Corollary 3.1. However, 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 the original 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.

By graph $G(V, E)$, we construct a new graph $G_i(V, E_i)$ such that an edge $(u, v) \in E_i$ if and only if $d(u, v) \leq 2^i$, $0 \leq i \leq \lceil \log n \rceil$. 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 5** *Given a connected undirected graph $G(V, E)$, the construction of $G_i(V, E_i)$ defined above can be done in $O(i)$ time using $O(n^3)$ processors, $0 \leq i \leq \lceil \log n \rceil$.*

**Proof** Let $G_0(V, E_0) = G(V, E)$. Assuming that $G_{i-1}$ already exists. Now we construct $G_i$ as follows: for any two vertices $u$ and $v$, we add to $E_i$ the edge $(u, v)$ if there exists at least one vertex $w$ ($w \neq u$ *and* $v$) such that edges $(u, w) \in E_{i-1}$ and $(v, w) \in E_{i-1}$. So, $G_i$ can be generated in $O(1)$ time using $O(n^3)$ processors if $G_{i-1}$ is given.□

**Lemma 6** *Let the graph $G_{\lfloor \log t \rfloor - 1}(V, E_{\lfloor \log t \rfloor - 1})$ be defined as above. Then for any two non-adjacent vertices $u$ and $v$, $d(u, v) > 2^{\lfloor \log t \rfloor - 1} \geq t/4$.*

The proof of Lemma 6 is so obvious, we omit it here. In the following we give an $NC$ algorithm for constructing a $2t$-spanner of $G$ with size $O(min\{m, n^2/t\})$.

**Algorithm 5**

Step 1. Construct the graph $G_{\lfloor \log t \rfloor - 1}(V, E_{\lfloor \log t \rfloor - 1})$.

Step 2. Find a maximal independent set $U_{\lfloor \log t \rfloor - 1}$ of graph $G_{\lfloor \log t \rfloor - 1}$ in parallel.

Step 3. For every $u \in U_{\lfloor \log t \rfloor - 1}$, construct a Breadth-First Search tree $T(u, \lfloor \log t \rfloor)$ rooted at $u$ with height $2^{\lfloor \log t \rfloor}$ in $G(V, E)$.

Step 4. The $2t$-spanner of $G$ is $\cup_{u \in U_{\lfloor \log t \rfloor - 1}} T(u, \lfloor \log t \rfloor)$.

**Lemma 7** $\frac{n}{\Delta^{\lfloor \log t \rfloor - 1} + 1} \leq |U_{\lfloor \log t \rfloor - 1}| \leq \frac{4n}{t}$.

**Proof** It is well known that, for a graph $G$ with the maximum degree $\Delta$ and the minimum degree $\delta$, the size of a maximal independent set $U$ of $G$ is in between $n/(\Delta + 1)$ and $n/\delta$. Whereas the the maximum degree and the minimum degree of graph $G_{\lfloor \log t \rfloor - 1}$ are $\Delta^{\lfloor \log t \rfloor - 1}$ and $\delta^{\lfloor \log t \rfloor - 1} \geq t/4$ respectively because there are no degree-one vertices in $G$. The lemma follows. □

Let $Cover(u, \lfloor \log t \rfloor)$ be a vertex 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 $V$.

**Lemma 8** *Let $v$ be a vertex belonging to covers $\mathcal{X}_1, \ldots, \mathcal{X}_s$, and $x_i$ be the center of $\mathcal{X}_i$, where $\mathcal{X}_i \in \mathcal{H}$ and $1 \leq s \leq i \leq |U_{\lfloor \log t \rfloor - 1}|$. Then there exists at most a partial cover $\mathcal{X}_j$ such that $d(v, x_j) \leq 2^{\lfloor \log t \rfloor - 2}$ and $d(v, x_i) > 2^{\lfloor \log t \rfloor - 2}$ for all other $\mathcal{X}_i$ with $i \neq j$.*

**Proof** If there exists two vertices $x_i$ and $x_j$ such that $d(v, x_i) \leq 2^{\lfloor \log t \rfloor - 2}$ and $d(v, x_j) \leq 2^{\lfloor \log t \rfloor - 2}$. Then $d(x_i, x_j) \leq d(v, x_i) + d(v, x_j) \leq 2^{\lfloor \log t \rfloor - 1}$. By the construction of the graph $G_{\lfloor \log t \rfloor - 1}$, there must have an edge $(x_i, x_j)$ connected these two vertices which contradicts the assumption that both $x_i$ and $x_j$ are included in the maximal independent set $U_{\lfloor \log t \rfloor - 1}$. □

**Lemma 9** *For any $u \in U_{\lfloor \log t \rfloor - 1}$, $|Cover(u, \lfloor \log t \rfloor)| \geq t/2$.*

**Proof** By the discussion in Lemma 7, the degree of vertex $u$ in $G_{\lfloor \log t \rfloor}$ is at least $t/2$.□

Recall that $Cover(u, \lfloor \log t \rfloor)$ is the home cover of vertex $v$ if $v \in Cover(u, \lfloor \log t \rfloor)$. Then we have

**Lemma 10** *Let $\mathcal{H}$ be defined as above. Then at least $t|U_{\lfloor \log t \rfloor - 1}|/8$ vertices has only one home cover.*

**Proof** We first show that the vertex set $U_{\lfloor \log t \rfloor - 1}$ is an independent set of graph $G_{\lfloor \log t \rfloor - 2}$. Without loss of generation, assuming that vertices $u,\ v \in U_{\lfloor \log t \rfloor - 1}$ are adjacent in graph $G_{\lfloor \log t \rfloor - 2}$. Then $d(u, v) \leq 2^{\lfloor \log t \rfloor - 2}$, so they must be adjacent in $G_{\lfloor \log t \rfloor - 1}$. This statement contradicts with the initial assumption that $U_{\lfloor \log t \rfloor - 1}$ is a maximal independent set of $G_{\lfloor \log t \rfloor - 1}$. Recall that $T(u, \lfloor \log t \rfloor - 2)$ is a Breadth-First Search tree rooted at $u$ with height $2^{\lfloor \log t \rfloor - 2}$ in $G(V, E)$. We denote by $Cover(u, \lfloor \log t \rfloor - 2)$ the set of all vertices in $T(u, \lfloor \log t \rfloor - 2)$. Obviously $Cover(u, \lfloor \log t \rfloor - 2) \subseteq Cover(u, \lfloor \log t \rfloor - 1)$. Now we prove that, for any $u$ and $v$ ($u,\ v \in U_{\lfloor \log t \rfloor - 1}$ and $u \neq v$), $Cover(u, \lfloor \log t \rfloor - 2) \cap Cover(v, \lfloor \log t \rfloor - 2) = \emptyset$. If not, there exists at least one vertex $w \in Cover(u, \lfloor \log t \rfloor - 2) \cap Cover(v, \lfloor \log t \rfloor - 2)$. Then $d(u, v) \leq d(v, w) + d(v, w) \leq 2^{\lfloor \log t \rfloor - 2} + 2^{\lfloor \log t \rfloor - 2} \leq 2^{\lfloor \log t \rfloor - 1}$. By the construction of the graph $G_{\lfloor \log t \rfloor - 1}$, $u$ and $v$ must be adjacent in this graph. Contradiction! While $|Cover(v, \lfloor \log t \rfloor - 2)| \geq t/8$ by the similar discussion as in Lemma 7 We therefore conclude that at least $t|U_{\lfloor \log t \rfloor - 1}|/8$ vertices belong to one home cover only. □

**Theorem 5** *The spanner generated by Algorithm 5 has size of $O(min\{m, n^2/t\})$ and factor of $2t$.*

**Proof** By Lemma 7, $|U_{\lfloor \log t \rfloor - 1}| \leq 4n/t$, and a vertex $v \in V$ belongs to at most $4n/t$ home covers. 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 $d(u,v) \leq 2^{\lfloor \log t \rfloor - 1}$, 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 6, 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 $d(u,v) \leq 2^{\lfloor \log t \rfloor}$. As a result, the distance between these two vertices in the spanner is no more than $2^{\lfloor \log t \rfloor} \leq t$. (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 $d(v,w) \leq 2^{\lfloor \log t \rfloor - 1}$ and $d(u,w) \leq 2^{\lfloor \log t \rfloor - 1}$, 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 $d(v,w) \leq d(w,u) + d(u,v) \leq 2^{\lfloor \log t \rfloor - 1} + 2^{\lfloor \log t \rfloor - 1} \leq 2^{\lfloor \log t \rfloor}$. (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 The algorithm for weighted graphs

Though the algorithm in [1] is very simple and efficient, it seems to be inherently sequential. Based on the *neighborhood cover* of Awerbuch and Peleg [6], Cohen [8] introduces the *pairwise cover* concept, and presents 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 $\epsilon'$ be any constant bounded by $0 < \epsilon' < 1/2$. Cohen's algorithm is described as follows.

**Algorithm 6**

Step 1. Initialization.

Step 1.1. $W_{max} := max\{wt(e) \mid e \in E\}$; $W_{min} := min\{wt(e) \mid e \in E\}$;
$R := \lceil \log_{1+\epsilon'}(\frac{W_{max}}{W_{min}}) \rceil$;

Step 1.2. **for** $i := 0$ to $R$ **do** in parallel
$W_i := W_{min}(1+\epsilon')^i$
**endfor**

Step 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**

Step 3. The spanner generated is $\cup_{i=1}^{r} \{T_i(u, \lfloor \log t \rfloor) \mid u \textit{ is the center of } \mathcal{X} \textit{ and } \mathcal{X} \in \mathcal{H}_i\}$.

The correctness of the above algorithm is testified in [8]. The key part of this algorithm is how to construct a coarse vertex cover on $V$ (called pairwise cover in

Cohen's algorithm) efficiently in parallel. As for this, Cohen [8] introduces randomness to chose the centers of covers. However, it seems not easy to transform her 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 function 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. Assuming $A^0 = A$. Then a coarse vertex cover $\mathcal{H}_i$ is constructed as follows.

**Algorithm 7**

Step 1. Construct an auxiliary graph $G_i(V, E^*)$. We add to $E^*$ the edge $(u, v)$ if and only if $d(u, v) \leq 2^{\lfloor \log t \rfloor - 1} W_i$.

Step 2. Find a maximal independent set $U(i, \lfloor \log t \rfloor - 1)$ of $G_i(V, E^*)$ in parallel.

Step 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 if and only if $d(u, v) \leq 2^{\lfloor \log t \rfloor} W_i$.

Step 4. The coarse vertex cover $\mathcal{H}_i = \{Cover_i(u, \lfloor \log t \rfloor) \mid u \in U(i, \lfloor \log t \rfloor - 1)\}$ is built.

**Lemma 11** *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 $d(u, v) \leq 2^{\lfloor \log t \rfloor - 1} W_i$ for fixed $i$, where $\mathcal{X} \in \mathcal{H}_i$.*

**Proof** By Algorithm 7, we know that 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 $O(\log^3 n)$ tine 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 $a^i_{(u,v)} \leq 2^{\lfloor \log t \rfloor} W_i$. 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 $|U(i, \lfloor \log t \rfloor - 1)| \leq n$ 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 6** *For a weighted connected graph $G(V, E)$ with non-negative weights, the spanner generated by Algorithm 6 and Algorithm 7 has size of $O(min\{m, \frac{n^2}{t} \log_{1+\epsilon}(\frac{W_{max}}{W_{min}})\})$ and factor of $2t$, where $\epsilon$ is a constant with $0 < \epsilon < 1/2$.*

**Proof** The proof is similar to Theorem 5. Omitted. □

Note that Theorem 6 holds only if the edges' weights of graph $G$ are independent of $n$. Besides that, we fail to make comparison of the weight of the spanner with $wt(MST)$.

## 5 Conclusions

In this paper, we first introduce the approximate $t$-spanner concept, and present simple parallel algorithms for finding such an approximate $t$-spanner in both unweighted graphs and weighted graphs. The algorithms exhibit a trade-off between the running time and the factor of the spanner generated. Then we present an NC algorithm for finding a $2t$-spanner on a weighted graph. It is very interesting to investigate whether the problem for finding a sparse $t$-spanner on both unweighted graphs and weighted graphs is in NC is still open.

## References


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

[2] S. Arya and M. Smid, Efficient construction of a bounded degree spanner with low weight, *ESA'94,* LNCS 855, 1994, pp.48-59.

[3] B. Awerbuch, Complexity of network synchronization, *J. Assoc. for Computing Machinery*, Vol. 32, No. 4, 1985, pp. 8054-823.

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

[5] B. Awerbuch and D. Peleg, Routing with polynomial communication-space trade-off, *SIAM J. Discrete Math.*, Vol. 5, No. 2, 1992, 151-162.

[6] B. Awerbuch and D. Peleg, Sparse partitions, *Proc. 31st IEEE Annual Sympo. on Foundations of Computer Science*, 1990, IEEE, 501-513.

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

[8] E. Cohen, Fast algorithms for constructing t-spanners and paths with t, *Proc. 34th IEEE Annual Sympo. on Foundations of Computer Science*, 1993, 648-658.

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

[10] G. Das and P. J. Heffernan, Constructing degree-3 spanners with other sparseness properties, *Interna. Sympo. on Algorithms and Computations*, Vol. LNCS 762, 1993, pp.11-20.

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

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

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

[14] D. Peleg and A. A. Schäffer, Graph spanners, *J. Graph Theory*, Vol. 13, No. 1, 1989, pp.99-116.

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

[16] D. Peleg and E. Upfal, A trade-off between space and efficiency for routing tables, *J. ACM*, Vol. 36, No. 3, 1989, 510-530.

[17] P. M. Vaidya, A sparse graph almost as good as the complete graph on points in K dimensions, *Discrete and Computational Geometry*, Vol. 6, 1991, 369-381.

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