Abstract—We consider optimal distributed computation of a given function of distributed data. The input (data) nodes and the sink node that receives the function form a connected network that is described by an undirected weighted network graph. The algorithm to compute the given function is described by a weighted directed acyclic graph and is called the computation graph. An embedding defines the computation communication sequence that obtains the function at the sink. Two kinds of optimal embeddings are sought, the embedding that: 1) minimizes delay in obtaining function at sink, and 2) minimizes cost of one instance of computation of function. This abstraction is motivated by three applications—in-network computation over sensor networks, operator placement in distributed databases, and module placement in distributed computing. We first show that obtaining minimum-delay and minimum-cost embeddings are both NP-complete problems and that cost minimization is actually MAX SNP-hard. Next, we consider specific forms of the computation graph for which polynomial-time solutions are possible. When the computation graph is a tree, a polynomial-time algorithm to obtain the minimum-delay embedding is described. Next, for the case when the function is described by a layered graph, we describe an algorithm that obtains the minimum-cost embedding in polynomial time. This algorithm can also be used to obtain an approximation for delay minimization. We then consider bounded freewidth computation graphs and give an algorithm to obtain the minimum-cost embedding in polynomial time.

Index Terms—Graph embedding, in-network function computation, operator placement.

I. INTRODUCTION

Efficient computing of functions of distributed data is of interest in many applications, most recently in sensor networks [1] and in programming models for distributed processing of large data sets, e.g., MapReduce [2] and Dryad [3].

Consider a typical sensor network scenario where sensor nodes measure and store environment variables and form a distributed database. These nodes also have computing and communication capabilities and can form a connected network. A sink node, also called terminal or receiver, is interested in one or more functions of this distributed data rather than in the raw data itself. Conventional examples for such functions are maximum, minimum, mean, parity, and histogram [1]. More sophisticated functions are also easily motivated, e.g., spatial and temporal correlations, spectral characteristics of the data that may be obtained by performing fast Fourier transform (FFT) on the data, and filtering operations on the raw data in sensor networks. A naive approach to obtain the required function(s) of the distributed data at the sink would be to collect the raw data at the sink and have it perform the computation. Alternatively, since the nodes have computation capability, it could possibly be more efficient to push the computation into the network, i.e., use a distributed computation scheme over the communication network. Our interest is in the latter approach—efficient “in-network computation” of the required function.

Several measures of efficiency of computation may be defined. Total energy expended in obtaining one sample of the function is a possible measure. Delay from the time at which the data is available at the sources to the time at which the function value is available at the sink is a second possible measure. If the data at each of the sources were to be a stream, then the rate at which the function values are available at the sink is a third possible measure. In this paper, we consider only the first two measures above—cost of computation and delay in computation. Thus, our focus is on algorithms that find an computation and communication (routing) sequence to compute a target function on a given network that minimize the delay or the cost. The target functions are assumed to belong to the class of functions that are computed using a scheme that can be represented as directed acyclic graphs (DAGs). The following example illustrates our intent.

Consider a network of $N$ nodes, $K$ of which collect measurement data from their environment. Let $x_k(t)$ be data sample available at node $k$ ($k = 1, \ldots, K$) at time $t$ and let $r(t) = \frac{1}{K} \sum_{i=1}^{K} x_i(t) x_{i+1}(t)$ be the function of interest. $r(t)$ can be computed using the schema of Fig. 1(a). Each edge in this directed graph represents an intermediate value in the computation of $r(t)$, and each node corresponds to an operation that is to be performed on its inputs. The communication network over which $r(t)$ is to be computed is shown in Fig. 1(b) as a weighted undirected graph. In the example, all the edges have unit weight. Two possible computation and communication schemes are shown in Fig. 1(c) and (d). We see that the scheme in Fig. 1(c) has a lower cost than that in Fig. 1(d).

Several flavors of in-network function computation exist in the literature. A randomly deployed multihop wireless network of noise free links is considered in [1] and [4]–[6]. They determine asymptotic achievable rates at which different symmetric functions like minimum, maximum, and type vector may be computed. An identical objective is addressed in [7]–[11] for
single-hop networks with noisy links and in [12] and [13] for multihop wireless networks with noisy links. When the nodes are in a \( \sqrt{n} \times \sqrt{n} \) grid and communicate over wireline or wireless links, [14] obtains the time and number of transmissions required to compute a function. Randomized gossip algorithms [15], [16], where a random sequence of node-pairs exchange data and perform a specific computation, are used in [17]–[19] for function computation. The interest is in time for all nodes to converge to the specified function. Another stream of work considers computing of specific functions using network coding and designs the communication networks to maximize the computation rate [20]–[22]. None of the above consider minimum cost function computation. Furthermore, to the best of our knowledge, only [23] considers computation of arbitrary functions in finite-size networks; the interest there is on maximizing the rate of computation.

Rather than arithmetic operations to perform a function computation, the nodes in Fig. 1(a) could correspond to operations required to execute a database query, and the directed edges would represent the flow of the results of the operations. In this case, the graph of Fig. 1(a) is called a query graph. The input data to this query graph could be from a distributed database, which in turn could be a sensor network; in this case, the graph in Fig. 1(b) would represent the interconnection among the units of the database and some other nodes that may be available for computation. In this context, performing an efficient query requires that the "operator placement" be efficient. Efficient operator placement is addressed in [24]–[29], all of which assume that the query graph is a tree. While [24], [27], and [29] develop heuristics-based algorithms for efficient operator placement when the query graph is a tree, algorithms with formal analyses are available in [25], [26], and [28]. Although [30] considers a non-tree query graph, only heuristic algorithms are provided.

Going back in the literature, we see that the "module placement" problem for distributed processing from the 1980s has a flavor similar to in-network computation and operator placement problems; see, e.g., [31]–[35]. In this case, the nodes of Fig. 1(a) correspond to the modules of a program, and a directed edge \((u, v)\) implies that module \(u\) calls module \(v\) during execution. Furthermore, the graph of Fig. 1(a) is called a call graph. The nodes of Fig. 1(b) represent the processors on which the modules of the program need to be executed and the edges represent the interprocessor links. The cost structure for this problem is more complex. The execution cost is specified for each module–processor pair, and there is also an interprocessor communication cost over an edge that in turn depends on the modules placed at the ends of the edge. The objective is to place the modules on the processors such that the total cost of execution is minimized. The first centralized algorithm for optimal module placement when the call graph is a directed tree was given in [32], and that for \(k\)-trees was given in [35]. Reference [31] showed that the problem can be efficiently solved for a two-processor system by using network flow algorithms, and [33] uses a similar approach to develop heuristic algorithms for a general call graph.

From the preceding discussion, we see that the objectives of, and hence the solution techniques for, efficient in-network computation, operator placement, and module placement all have a very similar theme—embedding (a formal definition is provided in Section II) a graph representing a computation schema on a connected weighted graph representing a network of processors. Much of the literature on such problems assumes that the graph describing the function is a tree. This is clearly very restrictive because the computation schema for a very large class of useful computable functions cannot be described by a tree and are more generally described by a DAG, e.g., FFT, sorting, any polynomial function of input data, and any function of Boolean data. MapReduce, the popular cloud computing paradigm, also has a DAG representation and is discussed in some detail in Example 3 in Section V.

**B. Organization of the Paper**

Our interest is in computing arbitrary functions that have a specific algorithmic representation and the communication network is an arbitrary network and not a random wireless network. (We reiterate that although we assume the in-network computation scenario to anchor the discussion, our results are also directly applicable to the operator and module placement scenarios.) Furthermore, we do not seek to specifically maximize the computation throughput; rather, our interest is in minimizing the cost or delay in a "one-shot" computation of the function. As we mentioned before, two measures of efficiency are used in this paper—cost of computation and delay in computation. Our results on minimum-cost embedding can be used to maximize computation throughput when used in the algorithms developed in [23].

Given an arbitrary function via its DAG description and an arbitrary network over which this function is to be computed, our first interest is to analyze the complexity of finding the optimal computation and communication scheme that will compute the function in the network. While much of the literature claims that this problem is NP-complete (for the case of minimizing cost), to the best of our knowledge a formal proof is not available; the literature eventually leads to a citation to a private communication in [32]. In Section III, we prove that, in general, both the minimum-cost and the minimum-delay embedding problems are NP-complete.

Some structure in the DAG can be exploited to provide polynomial-time algorithms for our problems. If the DAG is a tree, which is the assumption in much of the extant literature, the minimum-cost embedding is similar to shortest-path
algorithms [23], [28]. To the best of our knowledge, minimum-delay embeddings are not considered in the literature, and in Section IV, we provide a polynomial-time algorithm to find the minimum-delay embedding when the computation schema is a tree.

Next, we consider two large classes of computation graphs: 1) layered graphs; 2) bounded treewidth graphs. We derive the motivation for layered graphs from distributed data processing frameworks like MapReduce [2] and Dryad [3]. In Section V, we provide a polynomial-time algorithm to find the minimum-cost embedding when the DAG is a layered graph. This same algorithm also obtains an approximation for the minimum-delay embedding.

In Section VI, we show that the algorithm from Section V can find the minimum-cost embedding when the DAG is a bounded treewidth graph. An update mechanism when there is perturbation in the DAG is available in Appendix B. We conclude with a discussion in Section VII. The notation and the formal problem definition is described in Section II.

II. PRELIMINARIES

The communication network is represented by an undirected connected graph \( \mathcal{N} = (V, E) \) with \( V \) being the set of \( n \) nodes and \( E \) being the set of \( m \) edges. The elements of \( V \) are denoted by \( \{u_1, \ldots, u_n\} \). Each edge \( (u_i, u_j) \in E \) has a nonnegative weight \( T(u_i, u_j) \) associated with it. The weight could, for example, correspond to transmission time of a bit on the link, or the energy required to transmit one bit on the link or something more abstract. For a given \( T \), and any \( u_i, u_j \in V \), let \( d_{u_i, u_j} \) be the weight of the minimum-cost path from \( u_i \) to \( u_j \). Let \( |D - d_{u_i, u_j}| \) be the \( n \times n \) distance matrix. Of the \( n \) nodes in \( \mathcal{N} \), there are \( K \) source nodes denoted by \( \{s_1, s_2, \ldots, s_K\} \subset V \). Source node \( s_i \) generates data \( x_k \); denote \( x = \{x_1, \ldots, x_K\} \). A sink node \( t \in V \) requires to obtain a function \( f(x_1, x_2, \ldots, x_K) \) of the data.

We assume that schema to compute \( f(x_1, x_2, \ldots, x_K) \) is given and is represented by a directed acyclic graph \( \mathcal{G} = (\Omega, \Gamma) \), where \( \Omega \) is the set of \( p \) nodes and \( \Gamma \) is the set of \( q \) edges. The nodes in \( \mathcal{G} \) are denoted by \( \{\omega_1, \ldots, \omega_p\} \) and correspond to operations that need to be performed on the input data to the node, and the outgoing edges denote the flow of the result of these operations. Thus, each edge in \( \mathcal{G} \) represents a subfunction of the inputs. The sources in the computation graph are denoted by nodes \( \{\omega_1, \omega_2, \ldots, \omega_K\} \) with node \( \omega_i \) corresponding to source \( s_i \); node \( \omega_p \) is the sink that receives the function \( f(x_1, x_2, \ldots, x_K) \).

The direction on the edges in \( \mathcal{G} \) represent the direction of the flow of the data. Each edge \((\omega_i, \omega_j) \in \Gamma\) has a nonnegative weight \( w(\omega_i, \omega_j) \) associated with it, which could correspond to the number of bits used to represent the intermediate function.

Since \( \mathcal{G} \) is a directed acyclic graph, there is a partial order associated with its vertices. If \( (\omega_i, \omega_j) \in \Gamma \), then the function at \( \omega_j \) cannot be computed until the function at \( \omega_i \) is computed and the result forwarded to \( \omega_j \). Let \( \Phi_\Gamma(\omega) \) and \( \Phi_\Omega(\omega) \) denote, respectively, the immediate predecessors and successors of vertex \( \omega \), i.e., \( \Phi_\Gamma(\omega) = \{\tau \in \Omega | (\tau, \omega) \in \Gamma\} \) and \( \Phi_\Omega(\omega) = \{\tau \in \Omega | (\omega, \tau) \in \Gamma\} \). A processing cost (delay) function \( T : \Omega \times V \rightarrow \mathbb{R}^+ \) is used to capture the cost (delay) of performing a particular operation on a particular vertex of the network. Now we define the embedding of \( \mathcal{G} \) on \( \mathcal{N} \) as follows.

**Definition 1:** An embedding of a computation graph \( \mathcal{G} \) on a communication network \( \mathcal{N} \) is a many-to-one function \( E : \Omega \rightarrow V \) that satisfies the following conditions.

1) \( E(\omega_i) = s_i \) for \( i = \{1, \ldots, K\} \).
2) \( E(\omega_p) = t \).

In this definition of embedding, each node in the computation graph is mapped to a single node in the network graph, and the edge \((\omega_i, \omega_j) \in \Gamma \) is mapped to the shortest path between \( E(\omega_i) \) and \( E(\omega_j) \). Alternate definitions of an embedding are possible, e.g., an edge in \( \mathcal{G} \) can be mapped to more than one path in \( \mathcal{N} \) while satisfying some continuity constraints; this is the definition of an embedding in [23].

An embedding defines a computation and communication sequence in \( \mathcal{N} \) to obtain \( f \) at the sink. Let \( E \) be the set of all embeddings of \( \mathcal{G} \) in \( \mathcal{N} \) that follow Definition 1. The weight functions \( T \) and \( P \) can be treated as, respectively, the communication and the processing delays in \( \mathcal{N} \) for computing and communicating the subfunctions leading to computation of \( f \). We can define the delay in computing a subfunction by a node \( \omega \in \Omega \) in the embedding \( E \) as

\[
d(E(\omega_i)) := \max_{\omega_j \in E(\omega_i)} [d(E(\omega_j)) + w(\omega_j, \omega_i)d(E(\omega_j), E(\omega_i))] + P(\omega_i, E(\omega_i)).
\]

Recall that \( d_{u_i, u_j} \) is the length of the shortest path between vertices \( u_i, u_j \in V \); thus the first term here corresponds to the delay in obtaining all the operands at node \( E(\omega_i) \), and the second term corresponds to the processing delay at the node. Setting the delay at the sources to zero, i.e., \( d(E(\omega_i)) = 0 \) for all \( i \in [1, K] \), we can recursively calculate the delay of each vertices of \( \mathcal{G} \) on \( \mathcal{N} \). The delay \( d(E) \) of an embedding \( E \) is defined as the delay of the sink, i.e., \( d(E) := d(E(\omega_p)) \).

This leads us to the first problem that we consider in this paper: Find an embedding \( E_{\text{opt}} \) such that the delay of the embedding is minimum among all the embeddings for a given \( \mathcal{G}, \mathcal{N}, T, W \) and \( P \), i.e., solve the optimization problem

\[
E_{\text{opt}} := \arg \min_{E \in \mathcal{E}} \; d(E). \quad (\text{MinDelay})
\]

Alternatively, considering the weight functions \( T, W, \) and \( P \) as cost of communication and computation, e.g., the energy cost, the cost of an embedding can be defined as

\[
C(E) := \sum_{\omega \in \Omega} P(\omega, E(\omega)) + \sum_{(\omega_i, \omega_j) \in \Gamma} (w(\omega_i, \omega_j)d(E(\omega_i), E(\omega_j))).
\]

We can then find an embedding \( E_{\text{opt}} \) such that the cost of the embedding is minimum among all the embeddings for a given \( \mathcal{G}, \mathcal{N}, T, W \) and \( P \), i.e., solve the optimization problem

\[
E_{\text{opt}} := \arg \min_{E \in \mathcal{E}} \; C(E). \quad (\text{MinCost})
\]

The following example illustrates the preceding problems and the system setup.

**Example 1:** Consider a computation graph \( \mathcal{G} = (\Omega, \Gamma) \) and communication network \( \mathcal{N} = (V, E) \) shown in Fig. 2. The labels of each vertex in both the graphs are shown in Fig. 2. The processing cost (delay) for sources is assumed to be zero and for other vertices of \( \mathcal{G} \) it is assumed to be unity. An embedding \( E \) of \( \mathcal{G} \) on \( \mathcal{N} \) will have \( E(\omega_i) = s_i \); \( \forall i \in [1, 3] \) and \( E(\omega_p) = t \). Now consider two embeddings \( E_1, E_2 \) such that \( E_i(\omega_4) = a, E_i(\omega_5) = c, E_i(\omega_6) = d \) and
Using (2), it is easy to verify that $C(\mathcal{E}_1) = 36$ and $C(\mathcal{E}_2) = 34$. The delays in the embedding $\mathcal{E}_1$ are: $d(\mathcal{E}_1(\omega_1)) = \max(10, 2) + 1 = 11$, $d(\mathcal{E}_1(\omega_2)) = \max(8, 16) + 1 = 11$, $d(\mathcal{E}_1(\omega_3)) = \max(12, 12) + 1 = 13$, and finally $d(\mathcal{E}_1) = d(\mathcal{E}_1(\omega_4)) = 13 + 1 = 14$. Similarly, $d(\mathcal{E}_2) = 16$. Observe that the delay of $\mathcal{E}_1$ is lower among the two, but its cost is higher than that of $\mathcal{E}_2$.

Now we present an example that shows that the difference between the delay obtained by the solution of MinCost problem and that of the MinDelay problem can be of the order of the number of sources in the network.

Example 2: Consider the computation graph $G = (\Omega, \Gamma)$ and a network graph $\mathcal{N} = (V, E)$ as shown in Fig. 3 (a) and (b), respectively. The labels of the vertices are shown in the figure, and the numbers near the edges of $\mathcal{N}$ represent the weight of that edge. Note that the structure between $s_1$ to $u_2$ and $s_2$ to $v_2$ is repeated in the network graph $l$ times. Let us call this structure $A$. We assume that the weights on the edges of $\mathcal{G}$ are all one, and the processing costs are zero for sources, and for all other vertices it is assumed unity. Any embedding $\mathcal{E}$ of $\mathcal{G}$ on $\mathcal{N}$ will have $\mathcal{E}(\omega_i) = s_i \forall i \in [1, 2l]$ and $\mathcal{E}(\omega_i) - t$. Consider an embedding $\mathcal{E}_1$ such that $\mathcal{E}_1(\alpha_1) = u_1$, $\mathcal{E}_1(\alpha_2) = u_2$, $\mathcal{E}_1(\alpha_3) = v_2$, $\mathcal{E}_1(\alpha_3, 2) = u_{l+1}$. Let us compute the cost of this embedding. Note that the total cost of the embedding is $l$ times the cost coming from the structure $A$ plus the weight of edge $u_{l+1}$. The cost due to embedding of $A$ is the sum of the weights of edges $s_1 u_1, s_2 u_1, u_2 u_1, u_1 v_2$ and the processing costs at $u_1, u_2, v_2$. Hence, the cost of the embedding is $C(\mathcal{E}_1) = l(3 + 1.1a + 2e) + e$. Similarly, the delay of this embedding is $d(\mathcal{E}_1) = (a + e + 2)l + e$. Now consider another embedding $\mathcal{E}_2$ such that $\mathcal{E}_2(\alpha_1) = v_1$, $\mathcal{E}_2(\alpha_2) = u_2$, $\mathcal{E}_2(\alpha_3) = v_2$, $\mathcal{E}_2(\alpha_3, 2) = v_{l+1}$. The cost and delay for this embedding can be computed in a similar fashion to obtain $C(\mathcal{E}_2) = l(3 + 1.2a + 2e) + e$ and $d(\mathcal{E}_2) = l(0.7a + e + 2)l + e$. It can be shown that the first embedding $\mathcal{E}_1$ is the solution of MinCost, where as $\mathcal{E}_2$ is the solution of MinDelay problem of $\mathcal{G}$ on $\mathcal{N}$. If we use the solution of MinCost to get the solution of MinDelay, the difference would be $d(\mathcal{E}_1) - d(\mathcal{E}_2) = 0.3al$, which is of the order of the number of sources in the graph.

III. HARDNESS OF EMBEDDING

We begin by considering MinDelay. First consider the case when there is no processing delay, i.e., $P(\omega, E(\omega)) = 0$ in the network and the computation graph is unweighted, i.e., $\mathcal{w}(\omega_1, \omega_j) = 1 \forall (\omega_1, \omega_j) \in \Gamma$.

The delay of the embedding in this case is the delay of the longest embedded path from any source to the sink in $\mathcal{N}$. Let $d_{s,t}$ be the delay of the minimum-delay path between source $s_i$ and the sink $t$ in $\mathcal{N}$. Then, the delay of any embedding of $\mathcal{G}$ on $\mathcal{N}$ that follows the conditions of Definition 1 has to be more than the delay on the longest of all the minimum-delay paths from sources to sink. In other words, $d(\mathcal{E}) \geq \max(\mathcal{N}, 1, K)(d_{s,t})$. Now consider an embedding $\mathcal{E}^*$ that maps all the intermediate vertices of $\mathcal{G}$ to the sink in $\mathcal{N}$. The delay of this embedding will be $d(\mathcal{E}^*) = \max(\mathcal{N}, 1, K)(d_{s,t})$. Comparing it to $d(\mathcal{E})$ tells us that the embedding $\mathcal{E}^*$ minimizes the delay. Hence, MinDelay is easy to solve if there are no processing delays and $\mathcal{G}$ is unweighted.

Now we analyze the hardness of MinDelay and MinCost for arbitrary $\mathcal{G}$ and $\mathcal{N}$ and show that the both the optimization problems are NP-hard. We prove the hardness of the optimization problems by proving that the corresponding decision versions are NP-hard. The decision versions of the MinDelay and MinCost are defined as follows.

**Definition 2:** For a given $\mathcal{G}, \mathcal{N}, \Gamma, \mathcal{w}, \mathcal{P}$, and a positive number $L$, the decision version of the MinDelay problem outputs “yes” if there exists an embedding $\mathcal{E}$ of $\mathcal{G}$ on $\mathcal{N}$ such that $d(\mathcal{E}) \leq L$, and outputs “no” if no such embedding exists.

**Definition 3:** For a given $\mathcal{G}, \mathcal{N}, \Gamma, \mathcal{w}, \mathcal{P}$ and a positive number $L$, the decision version of the MinCost problem outputs “yes” if there exists an embedding $\mathcal{E}$ of $\mathcal{G}$ on $\mathcal{N}$ such that $C(\mathcal{E}) \leq L$, and outputs “no” if no such embedding exists.

Note that if one can solve the optimization version of MinDelay (resp. MinCost) problem in time, say $t$, then using the solution of that problem we can solve the corresponding decision version in time $O(t)$. Hence, the original optimization problem is at least as hard as its decision version. This implies that if we just prove that the decision version of the MinDelay (resp. MinCost) is NP-hard, then the optimization version is also NP-hard. We now proceed to prove that the decision version of our optimization problems are indeed NP-complete.
Theorem 1: The decision version of MinDelay problem is NP-complete.

Proof: We first prove that the decision version of MinDelay is NP-hard by giving a reduction from the NP-complete problem Precedence Constraint Scheduling with fixed mapping (PCS-FM) [36].

PCS-FM problem is defined as follows. Given a set of tasks $H$ with a partial order $\prec$ on it, task $h$ having length $l(h) = 1 \forall h \in H$, a set $\tau$ of $H$, and $m \in \mathbb{N}^+$ processors, find a schedule $\sigma$ of tasks on the processors that meets an overall deadline $D'$, maps each $h_i \in \tau$ to a particular processor $p(h_i)$, and obeys the precedence constraint that if $h_i < h_j$, then $\sigma(h_j) \geq \sigma(h_i) + 1$.

To the best of our knowledge, the hardness of PCS-FM problem has not been proved in the literature, and we provide the proof of NP-completeness of PCS-FM in Appendix A.

We first give a reduction from an instance $\phi = (H, \prec, \tau, p(\tau), m, l, D')$ of PCS-FM to an instance of MinDelay $\psi = (G, N', \gamma, \lambda, T, P, D)$, where $G = (\Omega, \Theta)$ and $N' = (V, E)$ are the computation and communication graphs, respectively. The set $\gamma \subseteq \Omega$ is to be mapped to $\lambda \subseteq V$ under any embedding $E$ and $T, P$ are the communication and processing delay of the network. Note that the set of tasks $H$ along with the partial order creates a DAG $\prec$, and we define $G = (\Omega, \Theta)$. We define $N'$ to be a complete graph on $m$ processors. Let $\gamma - \tau$ and $\lambda - p(\tau)$. The transmission delay $T$ between any two vertices of $N'$ is taken as $\epsilon = \frac{1}{\max T}$ and $P(\omega, u) = 1$ for all $\omega \in \Omega, u \in V$.

Finally, $D = D' + 1$.

We have to prove that there is a schedule $\sigma$ of $\phi$ that finishes in time $D'$ if and only if there is an embedding $E$ of $\psi$ with delay $D' \leq D < D' + 1$. The forward direction is easy to prove. If there is a schedule $\sigma$ of $\phi$ that finishes in $D'$ time and maps a task $h \in H$ to a processor $q$, then we can create an embedding of $\psi$ that maps the same vertex $h \in \Omega$ to a vertex $q \in V$. Note that because $\gamma - \tau$ and $\lambda - p(\tau)$, the conditions of Definition 1 are met in this embedding. The delay of any vertex $u \in G$ in this embedding will be at most the time at which the task $u$ finishes in the schedule $\sigma$ plus the number of times edges in $N'$ are used because of the same precedence order. Hence, the delay of this embedding will be $D' < D < D' + H|\epsilon| \leq D' + \frac{1}{\max T} < D' + 1$.

To complete the proof, we need to prove that if there is an embedding $E$ of $\psi$ with delay $\alpha \in \mathbb{N}^+$, then there is a schedule $\sigma$ of $\phi$ that finishes in time $\alpha$. We will create a schedule $\sigma$ from the embedding $E$. If a vertex $h \in \Omega$ is mapped to a vertex $u \in V$, then in the schedule $\sigma$ the task $h$ is executed by processor $u$. Because $\gamma = \tau$ and $\lambda = p(\tau)$, the tasks in $\tau$ are mapped to $p(\tau)$ in this schedule also. Let the total number of edges used in this embedding be $b$. Note that $b < |H|^2$ in any embedding where $H$ are the number of vertices in graph $G$. The total transmission delay in the embedding is $b e < 1$. Recall that the processing delays are all 1, hence the delay of the embedding can be written as $\alpha = |\alpha| + be$. It is easy to verify that the time required by the schedule $\sigma$ to complete is $|\alpha|$. This proves that the decision of MinDelay is NP-hard.

Given an instance of MinDelay problem, an embedding $E$ can be guessed nondeterministically and checked whether $d(E) \leq L$ in polynomial time. Thus, the decision version of MinDelay problem is in NP, and our reduction proves that it is in fact NP-complete.

We now look at the problem of approximating the MinDelay problem. We say that a polynomial-time approximation algorithm has a performance guarantee $\alpha > 1$ if it outputs a feasible solution of the problem that has delay at most $\alpha$ times the optimal solution. We prove the following.

Theorem 2: Unless P = NP, an instance of the MinDelay problem with $(G, N', T, P, D)$ and with unit processing delays and unit weights on the edges of $G$ does not have a polynomial-time approximation algorithm with performance guarantee strictly less than $100/99$ if its solution is greater than 10.

Proof: Note that while proving the hardness of MinDelay, we first reduced an instance of a PCS problem to an instance of a PCS-FM problem, and then we reduced PCS-FM to MinDelay problem. Let $opt_1$ and $sol_1$ be the deadline achieved by the optimal and a feasible solution of PCS problem, respectively. Similarly, let $opt_2$ and $sol_2$ be the deadline achieved by the optimal and a feasible solution of the PCS-FM problem. Finally, let $opt_3$ and $sol_3$ be the optimal and feasible solutions for MinDelay. While proving the NP-completeness of PCS-FM (in Appendix A), we showed that $opt_3 = opt_2 + 2$. We also showed that if a schedule of PCS problem achieves a deadline $sol_1$, then there is a schedule of PCS-FM with $sol_2 - sol_1 + 2$. Let $PCS-FM$ have a polynomial-time approximation algorithm with solution $sol_2 - x \cdot opt_2$. By [37, Observation 5.1], we know that the PCS problem does not have polynomial-time approximation algorithm with performance guarantee less than $1/3$. Hence, $sol_1 > \frac{3}{4} opt_1$, Substituting the relation between $sol_1$ and $sol_2$ in the above equation, we get $sol_2 - 2 > 3 opt_1$. Thus, $opt_2 > \frac{9}{2} opt_1 + 2$ and $x > \frac{9}{2} opt_1 + \frac{1}{2}$.

We now observe that $opt_1 > 1 \Rightarrow opt_2 > 3$, and hence $opt_2 > 1 \frac{3}{4} opt_1$ we can write $x > \frac{9}{2} opt_1 + 1$ and finally $x > 1 + \frac{1}{2}$. Hence, unless P = NP, PCS-FM cannot have a polynomial-time approximation algorithm with performance guarantee less than $100/99$. While proving the hardness for MinDelay, we showed that for any instance of PCS-FM with $sol_2 + 2$ we can get an instance of MinDelay problem with solution $sol_3$ such that $sol_3 < sol_2 + 2$. Let $MinDelay$ have a polynomial-time approximation algorithm with solution $sol_3 - y \cdot opt_3$. We know that $sol_3 > \frac{10}{9} opt_3$. Substituting the relation between $sol_3$ and $sol_2$ in the above expression, we get $sol_2 > \frac{3}{9} opt_3$ and $sol_2 = y opt_3 > \frac{10}{9} opt_2$. This implies that $y > \frac{10}{9} opt_3$. Observe that if $opt_2 > 10$, then $opt_2 > \frac{10}{9} opt_3 + 1$ (because of the reduction) implies that $opt_2 > \frac{1}{10} opt_3$. Thus, we get $y > 100/99$. This completes the proof.

We now consider the decision version of MinCost.

Remark 1: Recall that the cost of an embedding $E$ is computed using (2), which does not consider the direction on the edges of $G$. It only considers the weight $w$ on any edge of $G$. Thus, the cost of an embedding does not depend on the directions of the edges, and the solution of MinCost problem is same irrespective of whether the computation graph $G$ has directions or not.

Theorem 3: The decision version of the MinCost is NP-complete.

Proof: We actually prove that the decision version of MinCost is NP-hard even when the processing costs are zero and the costs on the edges of $G$ and $N$ are all one. In this case, the cost of the embedding $E$ is given by $C(E) := \sum w(u, v) \in E d_{u}(u, v, w_{u, v})$, where $d_{u, v}$ is the shortest distance between the vertices $u, v \in V$.

We prove the hardness of the decision version of MinCost by giving a reduction from the unweighted version of
**Multiterminal Cut** problem that is NP-complete [38]. Multiterminal Cut problem is defined as follows: Given an arbitrary graph \( G' = (V', E') \) and a set \( S = \{ s_1, s_2, \ldots, s_k \} \subseteq V' \) of \( k \) specified vertices, find the minimum number of edges \( E'_S \subseteq E' \) such that the removal of \( E'_S \) from \( E' \) disconnects each vertex in \( S \) from all the other vertices in \( S \).

The cost of an embedding can be computed in polynomial time using (2). Hence, given an instance of the decision version of the MinCost problem, one can guess an embedding in a nondeterministic way and check whether its cost is less than \( L \) or not in polynomial time. Thus, the decision version of the MinCost problem is in NP. To prove the NP-hardness of the problem, we will first show a transformation of an instance of Multiterminal Cut problem \( \psi = (N', S, D) \) to an instance of the decision version of MinCost \( \phi = (N, G, \gamma, \lambda, D) \). Then, we will show that there exists a set of edges in \( N' \) of size \( D \) that separates all the vertices of \( S \) from all other vertices in \( S \) if and only if there is an embedding \( \mathcal{E} \) such that a vertex \( \gamma_i \in \gamma \) is mapped to a vertex \( \lambda_i \in \lambda \) and has cost equal to \( D \).

From \( \psi \), define an instance of decision version of MinCost \( \phi \) as follows: Let \( N' \) be a complete graph on \( V' = \{ u_1, \ldots, u_k \} \) where \( k = |S| \) and \( G = N' \). Note that in this case \( G \) is undirected. Define \( \gamma = S \) and \( \lambda = V' \) such that a vertex \( \gamma_i \) is mapped to \( u_i \in V' \). In other words, \( k \) distinct vertices of \( \Omega \) are mapped to distinct vertices of \( V' \). Now we prove that there is an embedding \( \mathcal{E} \) of cost \( D \) if and only if \( \psi \) has a Multiterminal Cut of size equal to \( D \).

The “if” part is easy to see. If there is a minimum Multiterminal Cut \( E'_S \) of size \( D \) that divides the vertex set \( V' \) into \( k \) disjoint subsets, then define \( E \) such that each vertex in \( V'_i \) is mapped to \( u_i \in V' \). Then, the cost of the embedding is the total number of edges that go from \( V'_i \) to \( V'_j \) for \( i \neq j \). This is nothing but the size of the set \( E'_S \), which is equal to \( D \).

To complete the proof, we need to show that if there is no minimum Multiterminal Cut of \( \psi \) of size \( D \), then there is no embedding (which maps the vertices of \( \gamma \) to vertices of \( \lambda \)) of \( \phi \) with cost \( D \). Let us assume that there is no minimum Multiterminal Cut of \( \psi \) of size \( D \), but there is an embedding for \( \phi \) with cost \( D \). It implies that there is a mapping of \( \Omega \) on \( k \) different vertices of \( V \) such that \( \gamma_i \neq s_i \) is mapped to \( u_i \). Let us denote all the vertices of \( \Omega \) that are mapped to \( u_i \) by \( \Omega_i \). The cost of the embedding is equal to the number of edges between sets \( \Omega_i \) and \( \Omega_j \) for \( i \neq j \), which is equal to \( D \). Now it is easy to see that if we divide the vertices of \( N' \) in \( k \) disjoint subsets such that all the vertices in \( \Omega_i \) are in the same set, then we can create a Multiterminal Cut of the graph \( N' \) that has cost exactly \( D \). However, this contradicts our assumption. Hence, if there is an embedding \( \mathcal{E} \) for \( \phi \) of cost \( D \), then there is a Multiterminal Cut of \( \psi \) of same size. This proves that the decision version of MinCost is NP-hard when the computation graph is undirected. From Remark 1, thus the decision version of MinCost with directed computation graph is also NP-hard.

So far, we have not considered any weight functions. From [38], we know that the Multiterminal Cut problem for weighted graphs is also NP-complete. A simple modification in our reduction will prove that decision version of MinCost problem with weight functions is also NP-hard.

It is also shown in [38] that Multiterminal Cut problem is MAX SNP-hard. To prove any problem being MAX SNP-hard, it is sufficient to give a linear reduction to it from a known MAX SNP-hard problem. The linear reduction is defined as follows.

**Definition 4:** Let \( \Pi \) and \( \Pi' \) be two optimization problems. Then, we say that \( \Pi \) linearly reduces to \( \Pi' \) if there are two polynomial-time algorithms \( A, B \) and two constants \( \alpha, \beta > 0 \) such that the following.

1) Given an instance \( \pi \) of \( \Pi \) with an optimal cost \( \text{opt}(\pi) \), an algorithm \( A \) produces an instance \( \pi' = A(\pi) \) of \( \Pi' \) such that the cost of an optimal solution for \( \pi'(\text{opt}(\pi')) \) is at most \( \alpha \text{opt}(\pi) \), i.e., \( \text{opt}(\pi') \leq \alpha \text{opt}(\pi) \).

2) Given \( \pi, \pi' = A(\pi) \) and any solution \( y \) of \( \pi' \), there is an algorithm \( B \) that produces a solution \( x \) of \( \pi \) such that \( |\text{cost}(x) - \text{opt}(\pi)| \leq \beta \text{cost}(y) - \text{opt}(\pi') \).

Note that in the proof of Theorem 3, we use polynomial-time algorithms to reduce an instance \( \psi \) of the Multiterminal Cut problem to an instance \( \phi \) of the MinCost problem and we proved that \( \text{cost}(\psi) = \text{cost}(\phi) \). We can also get a solution of \( \phi \) from a solution of \( \psi \) and vice versa with parameters \( \alpha = \beta = 1 \). Hence, the reduction we used to prove that MinCost is NP-complete is in fact a linear reduction of Multiterminal Cut problem to MinCost. We thus have the following corollary.

**Corollary 1:** Problem MinCost is MAX SNP-hard and hence does not have any polynomial-time approximation scheme unless \( \text{P} = \text{NP} \) [39].

The NP-hardness of MinCost can also be proved by reduction from another well-known NP-complete problem \( k \)-clique [36]. The \( k \)-clique problem is defined as follows: Given an arbitrary graph \( N' \) and a positive integer \( k \), check whether \( N' = (V', E') \) has a clique (or a complete subgraph) of size \( k \). Since we know that the \( k \)-clique problem is \( \text{W}[1] \)-complete [40], the reduction from it also implies that MinCost does not have any fixed-parameter tractable algorithm and is also hard for \( \text{W}[1] \).

A variation of MinCost is considered in [35] in which the communication costs (computed as \( \mathcal{W}(\omega_i, \omega_j)\cdot d_{E(\omega_i)\cdot E(\omega_i)} \) in this paper) are not necessarily zero if \( E(\omega_i) = E(\omega_j) \). Furthermore, the source and the sink nodes are not fixed like in this paper.

The key complexity result is Lemma 2.1, which shows that their variation of MinCost with zero-one communication costs is NP-complete by reducing it from the planar 3-SAT problem.

With proof techniques do not allow the communication to be necessarily zero if \( E(\omega_i) = E(\omega_j) \). For non-zero-one communication costs, [35] also gives polynomial-time algorithms for their variation when \( G \) is a partial \( k \)-tree and almost trees with parameter \( k \).

As is the case with many NP-complete problems, our problems also become tractable when \( G \) has special structures. We consider three such structures that have wide applications—the tree, layered graphs, and bounded treewidth graphs. These are considered next.

**IV. \( G \) is a Tree**

Many functions that are useful on sensor networks, e.g., average, maximum, minimum, etc., can be represented by directed tree graphs. Operations that are required to resolve many database queries can also be represented as directed tree structures.

The trees representing a \( G \) is from a class of trees that have a set of leaf vertices whose in-degree is zero and a root vertex whose out-degree is zero. We consider a tree structured \( G \) such that all the leaf vertices represent the sources of data and the root acts as the sink that wants to know the final function value. Recall
Algorithm 1: Optimal embedding algorithm to solve MinDelay for tree graphs

Input: Network graph $\mathcal{G} = (V, E)$, $V = n$, $|E| = m$, Weight function $T : E \rightarrow \mathbb{R}^+$, Tree computation graph $G = (\Omega, \Gamma)$, $|\Omega| = p$, $|\Gamma| = p - 1$, Weight function $\omega : \Gamma \rightarrow \mathbb{R}^+$, Cost function $\Phi : \Omega \times V \rightarrow \mathbb{R}^+$.

Output: Embedding $E$ with minimum delay under $T$, $\omega$, and $\Phi$.

1: $[0 = d_{uv}]_n$ // $n \times n$ distance matrix for $\mathcal{G}$.

// Initialization of tables
2: $h_i(v) := 0$ $\forall v \in V, i \in [1, p - 1]$
3: $f_i(u, v) := 0$ $\forall u, v \in V, t \in [1, p - 1]$
4: for $i = 1$ to $p - 1$
  5:   for all $v \in V$
    6:     $f_i(s_i, v) := \omega_i + \Phi_i(\omega_i)d_{uv}$
    7:     $h_i(v) := f_i(s_i, v)$
    8:     $x_i(v) := s_i$
  9: end for
10: end for

// Backtracking
11: for $i = p - 1$ to 1 do
12:   for all $v \in V$
13:      for all $u \in V$
14:         $f_i(u, v) := \Phi_i(\omega_i) + f_i(u, v)$
15:      end for
16:      $h_i(v) := \min \{\max[h_i(u) | \omega_i \in \Phi_i(\omega_i)] + f_i(u, v)\}$
17:      $x_i(v) := \arg \min_u \{\max[h_i(u) | \omega_i \in \Phi_i(\omega_i)] + f_i(u, v)\}$
18:   end for
19: end for

// Algorithm Description
20: $d(E) := \max[h_i(t) | \omega_i \in \Phi_i(\omega_i)] + \Phi_i(\omega_i)$
21: $E(\omega_i) = t$

// Backtracking
22: for $i = p - 1$ to 1 do
23:   $E(\omega_i) = x_i(E(\Phi_i(\omega_i)))$
24: end for

that we label all the vertices in $\Omega$ as $\{\omega_1, \ldots, \omega_p\}$. Also $\Phi_i(\omega_i)$ and $\Phi_j(\omega_j)$ represent the predecessor and successor vertices of the vertex $\omega_i \in \Omega$. It is easy to verify that in this type of tree structured computation graph, every vertex (except the root) has only one successor vertex, i.e., for any $\omega_i \in \Omega$ (except the root) the set $\Phi_i(\omega_i)$ is a singleton set. The set $\Phi_i(\omega_i)$ is null, where $\omega_p$ is the root and there is a unique path from each source to the sink.

As we have mentioned earlier, [28] and [23] adapt, respectively, the Bellman–Ford and the Dijkstra shortest-path algorithms to solve MinCost when $G$ is a tree. In the following, we will describe Algorithm 1 that solves MinDelay when $G$ is a tree.

Algorithm Overview: Algorithm 1 is a centralized algorithm that assumes knowledge of the all-pair shortest path delay matrix $D$ of the communication network $\mathcal{N}$. The optimal embedding is computed by iterating through all the edges of the computation graph $G$. For an edge $(\omega_i, \omega_j)$ of $G$, the algorithm computes the optimal delay of the path leading to the vertex $\omega_j$ from sources via $\omega_i$ for all possible mappings of the vertex $\omega_j$ in the network. The delay until any vertex $\omega_j$ is the maximum of the optimal delays of all the paths reaching to $\omega_j$ plus the processing delay at that vertex. Once the optimal delay for the sink node is computed, the algorithm backtracks to find the optimal mapping of all the other vertices.

Algorithm Description: In each iteration, Algorithm 1 maintains the following data structures.

1) $f_i(u, v)$: It is the delay associated with edge $(\omega_i, \Phi_i(\omega_i))$ and vertex $\omega_i$ when $\omega_i$ and $\Phi_i(\omega_i)$ are mapped to vertex $u$ and $v$, respectively.

2) $h_i(v)$: It is the optimal delay of the path leading to vertex $\Phi_i(\omega_i)$ (via $\omega_i$) when it is mapped to $v \in \mathcal{N}$. The algorithm also stores the mapping of vertex $\omega_i$ in $x_i[v]$ corresponding to this value.

After initializing these data structures to zero (lines 2 and 3), the algorithm completes in the following three steps.

Lines 4–10: This is the iteration over all the sources in $\mathcal{G}$. As the mapping of source $\omega_i \in \mathcal{G}$ is fixed to source $s_i \in \mathcal{N}$, here we just calculate the minimum delay required to reach to all the vertices from the source $s_i$. Note that the processing delay at the source is zero, i.e., $\Phi(s_i, s_i) = 0$.

Lines 11–19: This is the main loop of the algorithm that runs over all the remaining vertices of $\mathcal{G}$ starting from $\omega_p \rightarrow s_i$ to $\omega_p \rightarrow s_1$. In each iteration, $f_i(u, v)$ is updated for all possible mappings of vertices $\omega_i, \Phi_i(\omega_i)$ (lines 13–15). The function $f_i(u, v)$ is computed by adding the following delay terms.

- $\Phi(s_i, s_i)$: This is the processing delay associated with the vertex $\omega_i$ when it is performed at vertex $u \in V$.

- $\omega_i(\omega_i) + \Phi_i(\omega_i)$: This is the communication delay associated with the edge $(\omega_i, \Phi_i(\omega_i))$ when mapped to $u$ and $v$, respectively.

Then, $h_i(v)$ is updated in line 16. Note that $\max[h_i(u) | \omega_i \in \Phi_i(\omega_i)]$ is the minimum delay until the vertex $\omega_i$ when it is mapped to $u$. This is equivalent to the first term in right-hand side of (1) (Section II). This along with the processing delay $\Phi(s_i, s_i)$ gives the delay of the vertex $\omega_i$ when mapped to $u$. As mentioned earlier, the algorithm also stores the mapping $u$ in $x_i[v]$, which minimizes the $h_i(v)$.

Lines 20–24: Once all the delays are computed, the algorithm computes the delay at the sink vertex $t$ (as the mapping of $\omega_p$ is fixed to $t$ in embedding $E$) and finds the mapping of vertices of $\mathcal{G}$ on $\mathcal{N}$, which gives this value by backtracking from the sink to the sources.

1) Analysis of Algorithm 1:

Theorem 4: Algorithm 1 solves MinDelay when $G$ is a tree and runs in $O(pn^2)$ time. Recall that $p$ and $n$ are the number of vertices in $G$ and $\mathcal{N}$, respectively.

Proof: We give the proof of correctness of Algorithm 1 only when the computation graph is unweighted. The proof can easily be extended to the case when there are weights on the edges of $G$. Recall that the delay of an embedding is defined recursively over all the vertices of $G$ starting from the sink vertex. It is sufficient to prove that at any iteration $i$, the algorithm computes the optimal delay of embedding the path from any source $\omega_i$ to an intermediate vertex $\Phi_i(\omega_i)$ via $\omega_i$ for all possible embeddings of $\Phi_i(\omega_i)$. Then at the end, it chooses the fixed mapping of the sink $\omega_p$ and traces back the optimal paths from sink to all the sources via the intermediate vertices. We will prove this inductively.

Let $\tilde{\omega}_i$ be the assignment of $\omega_i$ in $\mathcal{N}$ and at any iteration $\Phi_j(\omega_i) = \omega_j$ for some $j \in [1, p]$. The optimal path from any source $\omega_i$ for $i \in [1, K]$ to its successor vertex $\Phi_j(\omega_i)$ is just the shortest path distance between $s_i$ and the vertex to
which \( \Phi_j(\omega_i) \) will eventually be mapped. This is equal to \( d_{t_1, t_2} \). It is easy to verify that in the algorithm (line 7) this value is stored in \( d_{t_1} \) for all \( t \). Assuming that the optimal delays until the \( (t-1) \)th run are calculated by the algorithm and stored in \( d_{t_1}'s \), we will show that at \( t \)th run the algorithm computes the optimal delay. The optimal delay of the path from sources to \( v \) via \( u \) is given by \( d(\bar{x}_i) = \min_{\bar{v}} \{ \max_{\omega_i \in \Phi_j(\omega_i)} [d(\bar{x}_i) + d_{\bar{x}_i, \bar{v}_i}] + P(\omega_i, \bar{x}_i) \} \). Substituting value of \( d(\bar{x}_i) \) in \( f_i(\bar{u}_i) \), we get

\[
g_i(\bar{x}_j) = \min_{\bar{v}} \left\{ \max_{\omega_i \in \Phi_j(\omega_i)} g_i(\bar{x}_i) + P(\omega_i, \bar{x}_i) + d_{\bar{x}_i, \bar{v}_i} \right\}.
\]

Recall the line 14 of Algorithm 1 that computes \( f_i(\bar{u}_i) = \min_{\bar{v}} \{ \max_{\omega_i \in \Phi_j(\omega_i)} f_i(\bar{u}_i) + P(\omega_i, \bar{u}_i) \} \). This is nothing but the last two terms of the right-hand side of (3) when \( \bar{x}_i = u \) and \( \bar{x}_j = v \). Now observe line 16 of Algorithm 1, which computes \( h_i(\bar{u}_i) = \min_{\bar{v}} \{ \max_{v \in V} h_i(\bar{u}_i) \omega_i \in \Phi_j(\omega_i) + f_i(\bar{u}_i) \} \), where \( h_i(\bar{u}_i) \) is the optimal delay of the path leading to \( \omega_i \) via its predecessors \( \omega_i \). The optimal delay of the path \( h_i(\bar{u}_i) = g_i(\bar{x}_i) \), for \( \bar{x}_i = u \in V \). Hence, Algorithm 1 indeed computes the optimal delay of the path leading to \( \Phi_j(\omega_i) \) via \( \omega_i \) for all possible mappings \( v \in V \) of \( \Phi_j(\omega_i) \) and stores it in \( h_i(\bar{u}_i) \) at iteration \( t \).

Recall that \( \mathcal{G} \) is a tree, thus the total number of edges in it are \( p-1 \), and Algorithm 1 is executed once for each edge. In each iteration, it computes the delay of an edge for all possible mappings of its endpoints, which requires \( n^2 \) (line 14) time, where \( n \) is the number of vertices in \( \mathcal{N} \). Then, it adds the delay to the delay of its predecessors and chooses the one with minimum value \( h_i(\bar{u}_i) \) in line 16, which requires \( O(n) \) time. Hence, the time to complete one iteration is \( O(n^2 + n) = O(n^2) \). The total time complexity of Algorithm 1 is \( O(pn^2) \).

\section{V. \( \mathcal{G} \) Is A Layered Graph}

We now consider the case when \( \mathcal{G} \) is a layered graph. An example of a layered computation graph is shown in Fig. 4. We assume that there are \( r \) layers and number of vertices in each layer is at most \( k \). The vertices at layer \( l \) are labeled \( \omega_{l1}, \omega_{l2}, \ldots, \omega_{lk} \). The directed graph has edge \( (\omega_{ai}, \omega_{bj}) \) only if \( b = a \) or \( a + 1 \). Here, we also assume that all the sources are at layer one and there is only one sink \( t \) on the last layer.

We derive our motivation for this kind of computation graph from the MapReduce application framework. In MapReduce framework, each user comes to a network of processors with a set of Map and Reduce tasks. There is a precedence order between Map and Reduce tasks. Each Reduce task cannot be started unless the processing of corresponding set of Map tasks is finished. Each task takes predefined time to finish, and the outputs of the Map tasks are used by the corresponding Reduce tasks. This dependency can be represented by a directed graph with edges showing the dependency between the two tasks. The aim in this setting is to embed the Map and Reduce tasks on the processors such that the total time of computation and communication is minimized. We explain our motivation with the following example from [2].

\textbf{Example 3:} Consider a typical database query by a server (call it sink) to check the number of occurrences of different words in two large files that are available at two separate servers. The task of calculating the number of occurrences of each word inside the files can be divided into the following subtasks.

- **Splitting:** First the files are split into smaller subfiles such that each subfile can be processed by a processor in the network.
- **Mapping:** Each subfile is then parsed to get the number of times each word occurred in it.
- **Shuffling and reducing:** Once the count from each subfile is available, then the counts of one word are transported to one processor to compute the final count of that word. Each processor adds all the individual counts and results the final count of each word.
- **Final result:** Finally, the result is transported to the node that asked this query.

Fig. 5 represents a typical MapReduce data flow diagram for this problem. The aim is to determine the processors for each of the subtasks such that the time to answer the query at the sink is minimized. The whole process can be represented by a directed layered graph with each layer representing one subtask and a vertex at a layer representing a particular subtask. Observe that the edges in this graph are only between the consecutive layers and the operations at a vertex cannot start until the data from all its predecessors is available.

Now we present an algorithm that solves the MinCost problem for layered graphs in polynomial time.

\textbf{Algorithm Overview:} Like Algorithm 1, Algorithm 2 also has two phases—a forward path and a backward tracking. The forward path is a dynamic program that iterates over all the layers of the computation graph. In the first iteration, it computes the
Algorithm 2: Optimal embedding algorithm for layered graphs

**Input:** Network graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \), \( \mathcal{V} = n \), \( |\mathcal{E}| = m \), Weight function \( T : \mathcal{V} \rightarrow \mathbb{R}^+ \), Layered computation graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P}) \), \( \mathcal{V} = |\mathcal{V}| \), \( |\mathcal{E}| = |\mathcal{P}| = q \), Weight function \( \mathcal{W} : \mathcal{V} \rightarrow \mathbb{R}^+ \), Cost function \( \mathcal{P} : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}^+ \).

**Output:** Embedding \( \mathcal{E} \) with minimum cost

1: \( \mathcal{P} = \{p_{i,j} \}_{i,j} \) // \( n \times n \) distance matrix for \( \mathcal{N} \).
2: \( \mathcal{X} := \{X_1 : X_1 \subseteq \mathcal{V} \text{ and } |X_1| = k\} \) // |\( \mathcal{X} \)| = \( n^k \).
3: \( Y_k := \{Y_i : Y_i \subseteq \mathcal{V} \text{ and } |Y_i| = k\} \) // \( Y_k = n^k \).
4: \( z_{i,j} := \{z_{i,1}, z_{i,2}, \ldots , z_{i,k}\} \) \( i \in \mathcal{V} \) is the assignment of \( \omega_{i,j} \) in optimal embedding
5: \( h_{i,j} := \{h_{i,j} \}_{i=1}^{\mathcal{X}_i} \) \( \mathcal{X}_i \in \mathcal{X} \).
6: \( f_{i,j} := \{f_{i,j} \}_{i=1}^{\mathcal{X}_i} \) \( \mathcal{X}_i \in \mathcal{X} \).
7: \( x_{i,j} := \{x_{i,j} \}_{i=1}^{\mathcal{X}_i} \) \( \mathcal{X}_i \in \mathcal{X} \).

**Algorithm Description:** Algorithm 2 iterates over the layers in \( \mathcal{G} \) and in each iteration it maintains the following two data structures:
1) \( f_i(X_i, Y_j) \), the cost of embedding all the vertices until layer \( l \) when the vertices at layer \( (l + 1) \) are placed at \( Y_j \) and vertices of layer \( l \) are placed at \( X_i \), where \( X_i, Y_j \subset V \) of size \( k \);
2) \( h_i(Y_j) \), the optimal cost of embedding all the vertices (and corresponding edges) until layer \( l \) when the vertices at layer \( (l + 1) \) are mapped to an ordered subset \( Y_j \subset V \) of size \( k \).

After initializing these data structures to 0 (lines 5 and 6), the algorithm completes in the following three steps.

**Lines 7–15:** This is the main loop of the algorithm that runs for the first layer (from sources) to last but one layer. At each layer \( l \), the data structure \( f_i(X_i, Y_j) \) is updated for all possible combinations of \( k \) size subsets \( X_i \) and \( Y_j \) of \( V \) (lines 7–11). The following cost terms are added together to calculate \( f_i(X_i, Y_j) \) along with the optimal cost until layer \( l - 1 \), \( h_{i-1}(X_i) \).

- \( \sum_{a,b \in X_i} p_{i,a} \): Cost of putting computation node \( a \) at \( X_i \) for each node at the current layer.
- \( \sum_{a,b \in X_i, b \in Y_j} \mathcal{W}_{i,a} \mathcal{W}_{i+1,b} \): Total communication cost, when node \( \omega_{i,a} \) is placed at node \( a \) in \( V \) and \( \omega_{i+1,b} \) is placed at \( b \) in \( V \), is the multiplication of corresponding costs in computation graph (weight \( \mathcal{W} \)) and communication graph (weight \( \mathcal{W} \)). This term captures the cost for all the edges between layer \( l \) and layer \( l + 1 \).
- \( \sum_{a,b \in X_i, a \in Y_j} \mathcal{W}_{i+1,b} \mathcal{W}_{i,a} \): Total communication cost, when node \( \omega_{i,a} \) is placed at node \( a \) in \( V \) and \( \omega_{i,b} \) is placed at \( b \) in \( V \), is the multiplication of corresponding costs in computation graph (weight \( \mathcal{W} \)) and communication graph (weight \( \mathcal{W} \)). This term captures the cost for all the edges at layer \( l \).

**Lines 16–20:** Here, the algorithm finally computes the total cost of embedding the graph \( \mathcal{G} \) by adding the cost of the edges between the vertices of last layer \( r \), if any, when the vertices of the last layer are placed at \( X_r \). It computes the optimal cost of embedding \( \mathcal{E}, C(\mathcal{E}) \), by choosing the placement of the last layer that minimizes the overall cost (line 19–20). The vector \( \mathcal{Z}_r \) stores the mapping of vertices at layer \( r \) under the embedding \( \mathcal{E} \).

**Lines 21–23:** After finding the optimal mapping for the vertices at layer \( r \), the algorithm traces back the corresponding optimal mapping for vertices at layer \( r - 1 \) all the way up to the first layer.

To simplify the description of the algorithm, we do not show the fixed mapping of sources and sink into the network graph in Algorithm 2. It is easy to see that to incorporate the fixed mapping of sources and sink, the algorithm needs to only consider those subsets \( X_i, Y_j \) of \( V \) that map the source \( \omega_s \) (sink \( \nu \)) to the corresponding source \( s_i \) (sink \( t \)) while calculating the data structures of the layer on which \( \omega_i \) is present (layer \( r \)).

A. Analysis of Algorithm 2

**Theorem 5:** Algorithm 2 solves MinCost and the time complexity of the algorithm is \( O(n^{2r}k) \) when \( \mathcal{G} \) is a layered graph with \( r \) layers and at most \( k \) nodes per layer.
**Proof:** We prove the correctness of 2 when $G$ is an unweighted graph and the processing costs are all zero. We also assume that there are no edges between the vertices of a layer. The proof can easily be extended with weight functions $W$ and $P$. It is sufficient to show that at each iteration $i$, the algorithm computes the optimal cost of embedding the computation graph until layer $i$ for all possible embeddings of every node of layer $(i+1)$ given that the cost computed until layer $(i-1)$ is optimal. Let $\bar{x}_i$ be a vector of size $1 \times k$ of which its $i$th element represents the assignment of a network node for $w_i \in \Omega$. In other words, $\bar{x}_i$ is a $k$ size subset of $V$. Let us define

$$d(\bar{x}_i, \bar{x}_{i+1}) := \sum_{a \in \bar{x}_i, b \in \bar{x}_{i+1}} d_{a,b}.$$  \hspace{1cm} (4)

This represents the sum of distance between all the adjacent nodes in layer $i$ and $(i+1)$ when the nodes of layer $i$ are embedded to $\bar{x}_i$ and nodes of layer $(i+1)$ are embedded to $\bar{x}_{i+1}$. Total cost of any embedding can then be written as $C := \sum_{i=1}^{r-1} d(\bar{x}_i, \bar{x}_{i+1})$. To obtain the optimal embedding, we have to minimize the above equation with respect to all the possible mappings $\bar{x}_i$. Therefore, the optimal cost can be written as

$$C_{\text{opt}} = \min_{\bar{x}_1, \ldots, \bar{x}_r} \left( \sum_{i=1}^{r-1} d(\bar{x}_i, \bar{x}_{i+1}) \right).$$

Separating the terms with $\bar{x}_i$ and some algebraic manipulation will give us

$$C_{\text{opt}} = \min_{\tilde{x}_1, \ldots, \tilde{x}_r} \left( \sum_{i=1}^{r-1} \min_{\bar{x}_i} d(\tilde{x}_i, \bar{x}_{i+1}) \right).$$

where, $g(\tilde{x}_i)$ is the delay at layer $i$ from the source to $\tilde{x}_i$. Now it suffices to show that the algorithm indeed calculates $g_i$ at $i$th iteration. Recall that $\tilde{x}_i$ is a subset of $V$ that represents the mapping of nodes of layer $i$ and $(i+1)$, respectively. In the algorithm, $X_i$ represents a $k$ size subset of $V$ to which the nodes of the layer of the current iteration are mapped. In other words, $X_i$ is the same as $\tilde{x}_i$ of the above discussion. Similarly, $\bar{x}_i$ is the same as $\tilde{x}_{i+1}$. Note that in the first iteration, the algorithm calculates $f_i$ and $h_i$ as follows:

$$f_i(X_i, Y_j) = \sum_{a \in X_i, b \in Y_j} d_{a,b} + h_0(X_i)$$

for all $k$ size subsets $X_i$ and $Y_j$ of $V$. As $h_0(X_i)$ is initialized to zero for all $X_i$, using (4), we can write (7) as $f_i(X_i, Y_j) = d(X_i, Y_j) \quad \forall X_i \in X, Y_j \in Y$. Finally, $h_1$ is calculated by minimizing $f_i$ over $X_i$

$$h_1(Y_j) = \min_{X_i \in X} f_i(X_i, Y_j).$$

By comparing (5) and (8), we get $h_1(Y_j) = g_1(\tilde{x}_2)$, when $Y_j = \tilde{x}_2$. The algorithm maintains a table of $h_i$ and the value of $X_i$ for which $f_i(X_i, Y_j)$ is minimized for all the $k$ size subsets $Y_j$. This table is equivalent to storing the value of $g_1(\tilde{x}_2)$ for all possible values of $\tilde{x}_2$. Similarly at $i$th iteration the algorithm computes the following two terms:

$$f_i(X_i, Y_j) = d(X_i, Y_j) + h_{i-1}(X_i) \quad \forall X_i \in X, Y_j \in Y$$

$$h_i(Y_j) = \min_{X_i \in X} f_i(X_i, Y_j) \quad \forall Y_j \in Y.$$  \hspace{1cm} (9)

The algorithm stores the table of $h_i$ and corresponding $X_i$ for each $Y_j$. As $h_i(Y_j) = g_i(\tilde{x}_{i+1})$, the algorithm exactly calculates $g_i$ at each iteration and maintains a table for all possible embeddings for nodes at layer $i$ for each embedding of nodes at layer $i + 1$. This is the same as minimizing with respect to one $\bar{x}$ at a time as explained in (6). The computation of tables for $h_i$ only depends on the local variables, i.e., it only depends on the edges between layer $i$ and $(i+1)$ and all possible embeddings of nodes of layer $i$ and layer $(i+1)$.

As there are at most $k$ nodes at each layer, and there are $n$ possible locations where each node of computation graph can be placed in the communication graph, the time required to compute $f_i$ is $O(n^k \times k) = O(n^{2k})$ and the time to compute the corresponding $h_i$ is $O(n^k)$. Thus, total time to compute the table $h_i$ for a layer is $O(n^{2k} + n^k)$. There are $r$ layers in the computation graph, hence the computation of $h_i$ table is done at most $r$ time, which gives the time complexity of the algorithm as $O((n^{2k} + n^k) \times r) = O(rn^{2k})$.

We now claim that output of Algorithm 2 is a $k^2$ approximation of MinDelay. Let the cost obtained from the embedding $\mathcal{E}_c^d$ be $C_{\text{opt}}^d$. Once an embedding is given, we can obtain the delay of the embedding by recursively using (1) to find the delay at the sink. Let the delay of the embedding $\mathcal{E}_c^d$ be $D_{\text{opt}}^d$. Note that in finding the delay of any vertex in the embedding, we take the maximum of the delays coming from all its incoming edges, i.e., if the delays of the incoming edges are $d_1, d_2, \ldots, d_k$, then the delay at the vertex is $d = \max(d_1, \ldots, d_k)$. On the other hand, while computing the cost at any vertex, we add the costs coming from all its incoming edges, i.e., $c = d_1 + \cdots + d_k$. Hence, at any vertex $d \leq c$. This implies that for any embedding $\mathcal{E}$, $d(\mathcal{E}) \leq C(\mathcal{E})$. Between any two layers of a bounded width computation graph, there are at most $k^2$ edges, and if we assume that the delay on each edge is same, then the cost at any vertex is $c = k^2d$. With the same logic, one can easily prove that for an embedding $\mathcal{E}$, $C(\mathcal{E}) \leq k^2d(\mathcal{E})$. Thus, the minimum-cost embedding $\mathcal{E}_{\text{opt}}^d$ gives

$$\frac{C_{\text{opt}}^d}{k^2} \leq D_{\text{opt}}^d \leq C_{\text{opt}}^d.$$  \hspace{1cm} (10)

Let $\mathcal{E}_{\text{opt}}^d$ be the embedding that minimizes the delay of $G$ on $\mathcal{N}$ with $D_{\text{opt}}^d$ and $C_{\text{opt}}^d$ being its delay and cost, respectively. Then, we know that

$$\frac{C_{\text{opt}}^d}{k^2} \leq D_{\text{opt}}^d \leq C_{\text{opt}}^d.$$  \hspace{1cm} (11)

This implies that the cost of $\mathcal{E}_{\text{opt}}^d$ is a $k^2$ approximation of MinDelay when $G$ is a layered graph with $r$ layers and has at most $k$ nodes per layer.

VI. $G$ IS A BOUNDED TREEWIDTH GRAPH

We now extend the application of the algorithm of the preceding section to a graph that may not be a DAG. Toward that, we use the notion of the treewidth of the graph, which is a measure of how far the graph is from a tree. The following definition
of the treewidth of a graph is from [41] and reproduced here for the sake of completeness.

**Definition 5:** A tree decomposition of a graph \( G = (\Omega, \Gamma) \) is a tree \( T \) with vertices \( V_1, \ldots, V_r \) such that each \( V_i \subseteq \Omega \) and satisfies the following properties.

1) \( \cup_i V_i = \Omega \).
2) If \( u \in V_i \) and \( u \in V_j \), then \( u \in V_k \) for all \( V_k \) such that \( V_i, V_k, V_j \) form a connected component.
3) For all \( (u, v) \in \Gamma \), there exists a subset \( V_i \) such that both \( u, v \in V_i \).

The width of a tree decomposition is the size of largest \( V_i \) minus one. The treewidth \( tw \) of a graph is the minimum width among all possible tree decomposition of the graph.

In Section V, we presented Algorithm 2 to find the minimum-cost embedding of a layered graph when the edges are possible only between the consecutive layers. It is easy to observe that the treewidth of such a layered graph with maximum width \( k \) is \( tw = 2k - 1 \). The treewidth of the layered graph is shown in Fig. 6. A simple reinterpretation of the process of finding the minimum-cost embedding in Algorithm 2 gives us a procedure to find the minimum-cost embedding of the graphs with bounded (constant in terms of the size of the graph) treewidth.

Let us denote the vertices in the tree decomposition of the layered graph as \( V_1, \ldots, V_{r-1} \), where the vertex \( V_i \) contains all the vertices from layer \( i \) and \( (i+1) \). Observe that in the \( i \)-th run of the loop written in lines 9–11, Algorithm 2 computes the cost of embedding all the edges and nodes present in the vertex \( V_i \) of the tree decomposition for all possible mappings of nodes in \( V_i \).

In lines 12–13, the algorithm finds the optimal embedding cost of all the nodes and edges until \( V_i \) conditioned on the mapping of vertices in \( V_i \cup V_{i+1} \). Lines 16–18 compute the cost until the last layer, and then the algorithm traces back from \( V_{r-1} \) to \( V_1 \) to get the final optimal cost and the corresponding embedding (lines 19–23). Note that the time required to compute the cost until vertex \( V_i \) depends on the size of this vertex (lines 8 and 9 define that value). The total time to complete the process can be written as \( O(r tw + 1) \), where \( tw \) is the size of largest \( V_i \) in a tree decomposition.

Therefore, if we can find the tree decomposition of any computation graph \( G \) of \( r \) vertices with size of largest vertex to be \( tw \), then an algorithm similar to Algorithm 2 can be used to compute the minimum-cost embedding in time \( O(r tw + 1) \).

An example of a non-DAG graph for the technique discussed above is shown in Fig. 7(a). It is a computation schema that has a conditional jump, represented by the link between vertices \( e \) and \( a \). This is an example of a series-parallel graph [41], and such graphs are amenable to a tree decomposition cycle like in Fig. 7(b). In the preceding, we have shown that the technique used in Algorithm 2 can also be used to find the minimum-cost embedding of a series-parallel computation graph, among others, that can have a bounded tree decomposition.

### VII. DISCUSSION

#### A. Distributed Versions of the Algorithms

Note that Algorithms 1 and 2 are both centralized algorithms. In other words, they both need knowledge of \( N \). The algorithms have two stages with the all-pair distance matrix \( D \) of \( N \) computed in the first stage and the optimal value of \( h_i \) and \( f_{ij} \) using \( D \) computed in the second stage; see lines 6 and 14 of Algorithm 1 and lines 10 and 17 of Algorithm 2. Observe that once \( D \) is known, the outputs of our algorithms are independent of the vertex of the communication network that runs the algorithm. Hence, all vertices in \( N \) can run the algorithms to obtain the optimal embedding without interacting with the other vertices in the network. Hence, if one can compute the distance matrix in a distributed way in the network, then the optimal embedding can also be found in a distributed manner. Several distributed algorithms to find \( D \) are available, e.g., [42].

#### B. Delay in the Network With Bounded Capacity

In Section II, we discussed MinDelay, which finds the minimum-delay embedding of \( G \) on \( N \). The delay calculation in (1) (Section II) assumes that each edge \( l = (u_i, u_j) \in E \) has infinite capacity and can transmit as much data as needed in time \( T(l) \), where \( T(l) \) is the delay of an edge \( l \). In general, the edges in the network have finite capacity and can transmit only one type of data in time \( T(l) \). Here, we describe the delay in the network in the general setting. Recall that an edge \( \gamma \in \Gamma \) is mapped to a path in \( N \) in embedding \( E \) (defined in Definition 1). Delay of a network edge \( l \) is the time required for data to go from \( u_i \) to \( u_j \).

We say that an edge \( \gamma \in \Gamma \) has arrived at link \( l \) when the data corresponding to \( \gamma \) is ready for transmission on link \( l \) at vertex \( u_i \). Similarly, we say that the edge \( \gamma \) has departed from link \( l \) when the data reaches \( u_j \) via link \( l \). Let there be \( k \) edges \( \gamma_1, \ldots, \gamma_k \) of \( G \) mapped to an edge \( l \in E \) under embedding \( E \). Let the arrival time of these edges at the link be \( a_{l1}^{\gamma_1}, \ldots, a_{lk}^{\gamma_k} \). We assume that the capacity of each link is such that at a given time only one kind of data can be transmitted over it in the network. Then, the departure time of these edges from the link will be
Fig. 8. (a) Computation graph for function \( r(t) = (x_2 + x_3)(x_2 + x_4) \). (b) Communication network where every edge has delay 1.

\[ z^l_{\gamma_1} < z^l_{\gamma_2} \cdots < z^l_{\gamma_k} \]. The departure from the link \( l \) can be calculated recursively as follows: \( z^l_{\gamma_i} := \max(z^l_{\gamma_{i-1}}, a^l_{\gamma_i}) + T(l) \). Let an edge \( \gamma = (\omega_i, \omega_j) \in \Gamma \) be mapped to a path \( u_1, u_2, \ldots, u_m \) in \( \mathcal{N} \) under embedding \( \mathcal{E} \) such that \( \mathcal{E}(\omega_i) = u_1 \) and \( \mathcal{E}(\omega_j) = u_m \). Then, the delay of \( \gamma \) in the embedding is the sum of the delay incurred at each link \( u_1 u_2, u_2 u_3, \ldots, u_{m-1} u_m \), which can be written as

\[
d(\mathcal{E}(\gamma)) := d(\mathcal{E}(\omega_i), \mathcal{E}(\omega_j)) = z^{u_m-1}_{\gamma_m} - z^{u_1}_{\gamma_1} - a^u_{\gamma_1 u_2}.
\]

Now the delay of a vertex \( \omega_j \) in the embedding \( \mathcal{E} \) can be defined as (1), where \( d(\mathcal{E}(\omega_i), \mathcal{E}(\omega_j)) \) in computed as the above equation. We explain our point by an example.

**Example 4:** Consider a computation graph and a communication graph shown in Fig. 8. We consider that there are processing delays and delay associated with each edge of the communication graph is 1. Consider an embedding \( \mathcal{E} \) such that \( \mathcal{E}(a) = k, \mathcal{E}(b) = l, \mathcal{E}(c) = m \). Similarly, \( \mathcal{E}(s_1 a) = s_1 - k, \mathcal{E}(s_2 a) = s_2 - i - j - k, \mathcal{E}(s_3 b) = s_3 - i - j - l, \mathcal{E}(s_4 b) = s_4 - l, \mathcal{E}(ac) = k - m, \mathcal{E}(bc) = l - m \). It is easy to observe that using the delay model described in the Section II, the delay of the embedding is 5. While in the model described above as the embedding of edges \( s_2 a \) and \( s_3 b \) have a common edge \( i - j \) in them, the delay of the embedding increases to 6.

As mentioned in Example 4, the actual delay is more than the delay defined by (1) when multiple edges of \( \mathcal{G} \) are mapped to one edge of \( \mathcal{N} \) in an embedding. We study the impact of our assumption via simulations.

We studied the behavior of minimum-delay embedding and find the statistics on maximum number of times an edge of \( \mathcal{N} \) is used in the minimum-delay embedding of a typical \( \mathcal{G} \). In our study, the computation graph was taken to be a binary tree of \( p = 32 \) vertices, and its minimum-delay embedding on a random graph of \( n = 120 \) vertices was calculated using Algorithm 1. The probability of an edge being present in the random graph \( p_r \) was varied from 0.01 to 0.9. Note that as \( p_r \) increases, the number of edges in the network increases. The communication and transmission costs were assumed to be one, and the processing cost was chosen uniformly from integers [1, 10]. For each value of \( p_r \), 32 instances of network were generated, and for each instance Algorithm 1 was run for 10 random initial placements of sources and sink. The mean and the median of the maximum number of times an edge in the network graph is used in the embedding for each \( p_r \) is shown in Fig. 9. Observe that as the number of edges are increased in the network, maximum number of times an edge is used converges to one. Hence, we can say that for the networks with large number of edges compared to that of the computation graph, our assumption of delay calculated by (1) will be same as that of delay computed by (12).

**APPENDIX A**

**HARDNESS PCS-FM**

Here, we prove that PCS-FM problem is NP-complete by reducing it to another NP-complete problem, *Precedence Constraint Scheduling (PCS)* [36].

PCS problem is defined as follows. Given a set \( H' \) of tasks with a \(<\) partial order on it having length \( l(h) = 1 \) \( \forall h \in H' \) and \( m' \in \mathbb{Z}^+ \) processors, then find a schedule \( s \) of tasks on processors that meets an overall deadline \( D' \) and obeys the precedence constraints, i.e., if for some \( h, h' \in H' \), \( h < h' \), then \( \sigma(h) \geq \sigma(h') + 1 \).

First, we define an instance of PCS-FM problem \( \phi = (H, <, \tau, p(\tau), m, l, D') \) from an instance of PCS problem \( \psi = (H', <', m', l', D') \). We create partial order graph \( \leq \) from \( <' \) as shown in Fig. 10. The graph \( \leq \) has two parts: One part is the same as \( <' \), and the other part has \( k < m \) new vertices \( s_1, \ldots, s_k \) giving \( H = H' \cup \{s_1, \ldots, s_k\} \). The vertices \( s_1, \ldots, s_{k-1} \) are connected to \( s_k \) by a directed edge, and \( s_k \) is connected to all the vertices of \( <' \). Note that as all the edges are going away from \( s_k \), the new graph is still a DAG. Let \( m = m' \), and \( \tau = \{s_1, \ldots, s_k\} \). Define \( l(\bar{s}_i) = 1 \) \( \forall i \in [1, k] \) and \( l(h) = \bar{l}'(h) \) for all \( h \in H' \). Let us number the processors from 1 to \( m \) as \( q_1, \ldots, q_m \). Let us define \( p(s_i) := q_i \) and the deadline for \( \phi \) is \( D = D' + 2 \).

To prove our claim, we have to show that there exists a schedule \( s' \) for \( \psi \) that meets the deadline \( D' \) if and only if there exists a schedule \( s \) for \( \phi \) that meets the deadline \( D \). Now observe that in any schedule \( s \) for \( \phi \), any task in \( H' \) cannot start unless task \( s_k \) is finished, which in turn cannot start unless all the tasks \( s_1, \ldots, s_{k-1} \) are finished. As it is given that the tasks \( s_1, \ldots, s_{k-1} \) go to separate processors (due to \( p(s_i) \)), they can all be finished in 1 time-step giving \( s(s_k) = \sigma(s_k) + 1 = 1 \). Similarly, \( \sigma(h) \geq \sigma(s_k) + 1 = 2 \) for all \( h \in H' \). Hence, if there is a schedule \( s' \) for \( \psi \) that starts at 0 and finishes before
$D$, then a schedule $\sigma$ for $\phi$ can be defined as $\sigma(h) = \sigma'(h) + 2$
for all $h \in H$, $\sigma(s_i) = 0$ for all $i \in [1, k - 1]$ and $\sigma(s_k) = 1$. It is easy to observe that this is a valid possible schedule and finishes before $D' + 2$.

To complete the proof, we have to prove that if there is no schedule for $\psi$ that finishes before $D'$, then there is no schedule for $\phi$ that finishes before $D - D' + 2$. We prove this by contradiction. Let us assume that there is a schedule $\sigma$ for $\phi$ that finishes before $D$, but there is no schedule for $\psi$ that finishes before $D'$. As noted earlier, in any schedule for $\phi$, first two time-slots are required to finish tasks $s_1, \ldots, s_k$, and then only any other task can start. Hence in the schedule, $2 \leq \sigma(h') \leq D$ for all $h \in H'$. Total time taken to finish the tasks of $H'$ is $\leq D - 2 - D' + 2 = D'$. It means that there is a mapping of tasks of $H'$ on $m$ processors such that the total time to finish is less than $D'$, which implies that there is a schedule for $\psi$ that finishes before the deadline $D'$. This is a contradiction to our assumption. Hence, it is proved that the problem PCS-FM is as hard as PCS.

APPENDIX B
UPDATING SOLUTION TO $\text{MinCost}$ FOR PERTURBATIONS OF $G$

Let us consider a situation where the minimum-cost embedding for a layered graph $G$ is given and one needs to find the embedding for a new graph $G'$ that is generated by adding vertices and/or edges in $G$. We assume that $G'$ is still a layered graph with $r$ layers and maximum width $k$. Assume that we are given a set of $r$ tuples $(e_i, l_i)$ where edge $e_i$ is added at layer $l_i$. Note that edge $e_i$ should have at least one endpoint at the existing vertex in graph $G$. To find the new embedding, we first sort the $r$ tuples in $\{(e_1, l_1), \ldots, (e_r, l_r)\}$ such that $l_1 < l_2 < \cdots < l_r$. Now we start adding the edges layerwise from $l_1$ to $l_r$. At any layer $l$, the following three types of additions are possible.

1) Addition of a vertex with only one edge. When a vertex, say $v$, is added with an edge $uw$ to an existing vertex $u$ at layer $l$, then $v$ can be seen as a sink to an intermediate function value available at $u$. Let us assume that the vertex $u$ is mapped to vertex $z_u \in V$ under the original embedding. If the mapping of $z_u \in V$ of $v$ is predefined (which is generally the case for all the sources and sinks in the network), then we just have to find the minimum-cost path between $z_u$ and $z_v$ and add it to the existing embedding to find the new embedding. If the mapping of $v$ is not predefined, then mapping it to $z_v$ will give the minimum-cost embedding. This can be done in $O(1)$ time.

2) Addition of an edge between two existing vertices. Let us consider a situation when an edge $uv$ is added such that $u$ is at layer $l$ and $v$ is at layer $(l + 1)$ with weight $w(u, v)$. The data structure already available for each layer $l$ after running Algorithm 2 is shown in Fig. 11. Recall that $f_l(X_i, Y_j)$ is the cost of embedding until layer $l$ (including the edges between layer $l$ and $(l + 1)$) when vertices of layer $l$ are at $X_i$ and that of $(l + 1)$ are at $Y_j$. $h_l(Y_j)$ is the optimal cost until layer $l$ when vertices at layer $(l + 1)$ are at $Y_j$. If the vertex $u$ is placed at $x_u \in X_i$ and $v$ is placed at $y_v \in Y_j$, then the new cost of embedding is $f_l'(X_i, Y_j) = f_l(X_i, Y_j) + w(u, v)$. This line number 10 of Algorithm 2). One needs to modify the whole $f_l$ matrix at layer $l$ by adding a value, and correspondingly the minimum cost $h_l(Y_j)$ will also change (lines 12 and 13). As

the value of $h_l(X_i)$ changes the pointers to compute $f_l$, the values for all the subsequent layers will also change. Hence at each layer $l$, we will modify the whole data structure just by adding new values of $h_{l-1}$ and subtracting the old values of $h_{l-1}$. Once all the data structures are changed, the algorithm needs to run the same backtracking procedure (lines 19–23) to get the new embedding. Assuming that the modification in the data structure at each layer can be done in $O(T)$ time, then the new embedding can be found in $O(rT)$ time.

3) Addition of a vertex with more than one edge. Let a vertex $v$ be added to layer $l$ with more than one edges to existing vertices at layer $(l - 1)$ and/or $(l + 1)$. The width of the layer $l$ is still upper-bounded by $k$. By following the same logic presented above, it is easy to observe that now the data structures from layer $(l - 1)$ onward will change, i.e., $f_l-1(X_i, Y_j)$ and $h_l-1(Y_j)$ will also change. The new embedding can again be found in $O(rT)$ time. Note that at a layer $l$ the data structure changes due to edges added to layers $1, \ldots, (l - 1)$ (which takes $O(l - 1)$ time) and the edge added at layer $l$ (which takes only $O(T)$ time as described earlier). Hence the total time to change all the data structures in this process will be just $O(rT)$, where $r$ is the total number of layers, as opposed to $O(r^2T)$ if we add each edge separately.

REFERENCES

[1] A. Giridhar and P. R. Kumar, “Computing and communicating functions over sensor networks,” IEEE J. Sel. Areas Commun., vol. 23, no. 4, pp. 755–764, Apr. 2005.

[2] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in Proc. 6th OSDI, 2004, pp. 137–150.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterli, “Dryad: Distributed data-parallel programs from sequential building blocks,” in Proc. EuroSys, 2007, pp. 59–72.

[4] A. Giridhar and P. R. Kumar, “Toward a theory of in-network computation in wireless sensor networks,” IEEE Commun. Mag., vol. 44, no. 4, pp. 98–107, Apr. 2006.

[5] N. Khude, A. Kumar, and A. Karnik, “Time and energy complexity of distributed computation in wireless sensor networks,” in Proc. IEEE INFOCOM, 2005, pp. 2625–2637.

[6] S. Kamath, D. Manjunath, and R. Mazumdar, “On distributed function computation in structure-free random networks,” in Proc. IEEE INFOCOM, 2005, pp. 2625–2637.

[7] S. Kamath, D. Manjunath, and R. Mazumdar, “On distributed function computation in structure-free random networks,” in Proc. IEEE Inf. Theory, vol. 60, no. 1, pp. 432–442, Jan. 2014.

[8] R. G. Gallager, “Finding parity in simple broadcast networks,” IEEE Trans. Inf. Theory, vol. 34, no. 2, pp. 176–180, Mar. 1988.

[9] E. Kushilevitz and Y. Mansour, “Computation in noisy radio networks,” in Proc. SODA, 1998, pp. 236–243.

[10] U. Feige and J. Kilian, “Finding OR in noisy broadcast networks,” Inf. Process. Lett., vol. 73, no. 1–2, pp. 69–75, Jan. 2000.

[11] I. Newman, “Computing in fault tolerance broadcast networks,” in Proc. 19th IEEE Annu. Conf. Comput. Complexity, 2004, pp. 113–122.
[11] N. Goyal, G. Kindler, and M. E. Saks, “Lower bounds for the noisy broadcast problem,” in Proc. IEEE FOCS, Oct. 2005, pp. 40–52.

[12] L. Ying, R. Srikant, and G. Dullerud, “Distributed symmetric function computation in noisy wireless sensor networks,” IEEE Trans. Inf. Theory, vol. 53, no. 12, pp. 4826–4833, Dec. 2007.

[13] C. Dutta, Y. Kanoria, D. Manjunath, and J. Radhakrishnan, “A tight lower bound for parity in noisy communication networks,” in Proc. 19th Annu. ACM-SIAM SODA, San Francisco, CA, USA, 2008, pp. 1056–1065.

[14] N. Karamchandani, R. Appuswamy, and M. Franceschetti, “Time and energy complexity of computation over networks,” IEEE Trans. Inf. Theory, vol. 57, no. 12, pp. 7671–7684, Dec. 2011.

[15] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, “Gossip algorithms: Design, analysis and applications,” in Proc. IEEE INFOCOM, Miami, FL, USA, 2005, pp. 1653–1664.

[16] D. Shah, “Gossip algorithms,” Found. Trends Netw., vol. 3, no. 1, pp. 1–125, 2009.

[17] D. Mosk-Aoyama and D. Shah, “Computing separable functions via gossip,” in Proc. 22nd ACM PODC, 2006, pp. 113–122.

[18] O. Ayaso, D. Shah, and M. A. Dahleh, “Counting bits for distributed function computation,” in Proc. IEEE ISIT, 2008, pp. 652–656.

[19] S. Bodas and D. Shah, “Fast averaging,” in Proc. IEEE ISIT, 2011.

[20] A. Ramamoorthy and M. Langberg, “Communicating the sum of sources over a network,” 2010 [Online]. Available: http://arxiv.org/abs/1011.5319

[21] R. Appuswamy, M. Franceschetti, N. Karamchandani, and K. Zeger, “Network coding for computing: Cut-set bounds,” IEEE Trans. Inf. Theory, vol. 50, no. 2, pp. 1015–1030, Feb. 2011.

[22] B. K. Rai and B. K. Dey, “On network coding for sum-networks,” IEEE Trans. Inf. Theory, vol. 58, no. 1, pp. 50–63, Jan. 2012.

[23] V. Shah, B. K. Dey, and D. Manjunath, “Network flows for function computation,” IEEE J. Sel. Areas Commun., vol. 31, no. 4, pp. 714–730, Apr. 2013.

[24] B. Bonfils and P. Bonnet, “Adaptive and decentralized operator placement for in-network query processing,” Telecommun. Syst., vol. 26, pp. 389–409, 2004.

[25] U. Srivastava, K. Munagala, and J. Widom, “Operator placement for in-network stream processing,” in Proc. 24th ACM SIGMOD-SIGACT-SIGART Symp. Principles Database System, 2005, pp. 250–258.

[26] Z. Abrams and J. Liu, “Greedy is good: On service tree placement for in-network stream processing,” MSR, Tech. Rep., 2005.

[27] P. Pietzuch et al., “Network-aware operator placement for stream-processing systems,” in Proc. Int. Conf. Data Eng., Atlanta, GA, USA, 2006, p. 49.

[28] L. Ying, Z. Liu, D. Towsley, and C. H. Xia, “Distributed operator placement and data caching in large-scale sensor networks,” in Proc. IEEE INFOCOM, 2008, pp. 1651–1659.

[29] Z. Lu, Y. Wen, R. Fan, S. L. Tan, and J. J. Biswas, “Toward efficient distributed algorithms for in-network binary operator tree placement in wireless sensor networks,” IEEE J. Sel. Areas Commun., vol. 31, no. 4, pp. 743–755, Apr. 2013.

[30] A. Phtak and V. K. Prasanna, “Energy-efficient task mapping for data-driven sensor network macroprogramming,” IEEE Trans. Comput., vol. 59, no. 7, pp. 955–968, Jul. 2010.

[31] H. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” IEEE Trans. Softw. Eng., vol. SE-3, no. 1, pp. 85–93, Jan. 1977.

[32] S. Bokhari, “A shortest tree algorithm for optimal assignments across space and time in a distributed processor system,” IEEE Trans. Softw. Eng., vol. SE-7, no. 6, pp. 583–589, Nov. 1981.

[33] V. Mary Lo, “Heuristic algorithms for task assignment in distributed systems,” IEEE Trans. Comput., vol. 37, no. 11, pp. 1384–1397, Nov. 1988.

[34] S. Bokhari, “Partitioning problems in parallel, pipelined, and distributed computing,” IEEE Trans. Comput., vol. 37, no. 1, pp. 48–57, Jan. 1988.

[35] D. Fernandez-Baca, “Allocating modules to processors in a distributed system,” IEEE Trans. Softw. Eng., vol. 15, no. 11, pp. 1427–1436, Nov. 1989.

[36] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco, CA, USA: Freeman, 1979.

[37] J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger, “Non-approximability results for scheduling problems with minsum criteria,” in Integer Programming and Combinatorial Optimization. Berlin, Germany: Springer, 1998, Lecture Notes in Comput. Sci.

[38] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, “The complexity of multiterminal cuts,” SIAM J. Comput., vol. 23, pp. 864–894, 1994.

[39] S. Arora, C. Lund, R. Motwani, M. Sudan, and S. Szegedy, “Proof verification and intractability of approximation problems,” in Proc. IEEE Symp. Found. Comput. Sci., 1992, pp. 13–22.

[40] R. G. Downey, Parameterized Complexity. New York, NY, USA: Springer-Verlag, 1999.

[41] R. Diestel, Graph Theory. New York, NY, USA: Springer-Verlag, 2000.

[42] S. Kanchi and D. Vineyard, “An optimal distributed algorithm for all-pairs shortest-path,” Inf. Theories Appl., vol. 11, pp. 141–146, 2004.

Pooja Vyavahare received the B.E. degree in electronics and telecommunication from Shri Govindram Seksaria Institute of Technology and Science, Indore, India, in 2005, and the M.Tech. degree in communications engineering from the Indian Institute of Technology (IIT) Delhi, New Delhi, India, in 2007, and is currently pursuing the Ph.D. degree in electrical engineering at IIT Bombay, Mumbai, India. From 2007 to 2009, she was with Texas Instruments India Pvt. Ltd., Bangalore, India, as a Software Engineer. From 2009 to 2011, she was Lecturer with the Department of Electronics Engineering, K. J. Somaiya College of Engineering, University of Mumbai, Mumbai, India. Her research interests include analysis of communication networks, algorithm design, and optimization.

Miss Vyavahare is a recipient of the IEEE SPCOM 2014 Best Student Paper Award and winner of the University Gold Medal Award in 2005.

Nutan Limaye received the Ph.D. degree in theoretical computer science from the Institute of Mathematical Sciences, Chennai, India, in 2009.

She was a Visiting Fellow with the Tata Institute of Fundamental Research (TIFR), Mumbai, India, from 2009 to 2010. She has been an Assistant Professor with the Computer Science and Engineering Department, Indian Institute of Technology (IIT) Bombay, Mumbai, India, since 2010. Her primary research interests lie in complexity theory and algorithms. Her recent research has been on arithmetic circuit lower bounds, design of small depth proof systems, and design of space-efficient algorithms.

D. Manjunath (S’86–M’93–SM’13) received the B.E. degree in electronics and communication engineering from Mysore University, Mysore, India, in 1986, the M.S. degree in electrical engineering from the Indian Institute of Technology (IIT) Madras, Madras, India, in 1989, and the Ph.D. degree in electrical engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1993.

He has been with the Electrical Engineering Department, IIT Bombay, Mumbai, India, since 1998, where he is now an Institute Chair Professor. He has previously worked with the Corporate R&D Center, General Electric, Schenectady NY, USA, in 1990; the Computer and Information Sciences Department, University of Delaware, Newark, DE, USA, from 1992 to 1993; the Computer Science Department, University of Toronto, Toronto, ON, Canada, from 1993 to 1994; and the Department of Electrical Engineering, IIT Kanpur, Kanpur, India, from 1994 to 1998. At IIT Bombay, he has been the Head of the Computer Centre since 2011. He is a coauthor of two textbooks, Communication Networking: An Analytical Approach (Morgan-Kaufman, 2004) and Wireless Networking (Morgan-Kaufmann, 2000). His research interests are in the general areas of communication networks and performance analysis. His recent research has concentrated on random networks with applications in wireless and sensor networks, network pricing, and queue control.

Prof. Manjunath is an Associate Editor of the IEEE TRANSACTIONS ON NETWORKING, Queueing Systems: Theory and Applications, and Sadhana: The Proceedings of the Indian Academy of Sciences. He was TPC Chair for COMSNETS 2011 and 2013, and General Chair for ACM MobiHoc 2013 and COMSNETS 2013. He is a recipient of the Best Paper Award at ACM SIGMETRICS 2010.