Efficient Flow Allocation Algorithms for In-Network Function Computation

Virag Shah Bikash Kumar Dey D. Manjunath

***Abstract*—We consider in-network computation of an arbitrary function over an arbitrary communication network. We consider the same model as our earlier work in [1]. The given network consists of directed/undirected links with capacity constraints, some source nodes which generate data, and other intermediate forwarding nodes. An arbitrary function of this distributed data is to be computed at a terminal node. The structure of the function is described by a given computation schema represented by a directed tree. In our earlier work, we presented a linear program to define the maximum rate of computation, and then introduced a notion of flow-conservation suitable in this context so as to come up with a flow-conservation based linear program which can be solved in polynomial time. In this paper, we develop a combinatorial primal-dual algorithm to obtain $(1-\epsilon)$-approximate solutions to these linear programs. As a part of this, we present an algorithm to find a minimum-cost embedding in a network of weighted links—which is of independent interest. We then describe application of our techniques to other practically interesting problems of multiple terminals wanting different functions, computation with a given desired precision, and to networks with energy constraints at nodes.**

I. Introduction

In-network function computation, or more expansively, distributed computation of a function of a distributed data vector, is an important performance enhancing mechanism for sensor networks and other similar applications. Two characteristics are usually exploited: (1) the network wants a a function, say Θ, of the distributed data vector and not the actual data vector, and (2) network nodes can perform computation, in addition to communication, and can therefore possibly participate in a distributed algorithm. For a sensor network like application it is reasonable to assume that the data vector, e.g., the measurement values at the sensors, could be performed at any rate and the limitation is only in the form of the rate at which the desired function can be computed. Thus we will assume that the data vector—the argument of Θ—is a vector time series that can be generated at any rate; equivalently, an infinite sequence of the vector is available. And we want to compute Θ at the maximum possible rate.

In-network computation has several facets and there is some literature addressing many of these facets. The information theoretic literature considers simplistic fixed networks with a small number of correlated sources, e.g. [2], [3]. The algorithms and complexity theory literature is interested in the communication complexity of the functions in specific communication scenarios; [4] is an excellent introduction and, more recently, [5] considers a new problem in a more complex setting. Along similar lines, the number of transmissions required to compute a function of one bit data at n nodes in a broadcast network with link errors is the subject of interest in [6] and several follow up works. A stochastic geometry view was laid down in [7] and has had several follow up works, e.g., [8]. There is also a significant interest in the network coding community on distributed function computation over fixed arbitrary networks with independent sources. However, designing optimal coding schemes and finding capacity is a difficult problem except for very special functions or networks [9], [10].

This work was supported in part by the Department of Science and Technology, India, and the Bharti Centre for Communication, IIT Bombay, India. V. Shah is with Department of ECE, University of Texas at Austin. He was with IIT Bombay during the course of the work. B. K. Dey and D. Manjunath are with Department of Electrical Engineering, IIT Bombay. Emails: virag4u@gmail.com,{bikash,dmanju}@ee.iitb.ac.in

A. A Motivational Example and Preview of Results

In this paper we consider a significantly different approach from those in the preceding discussion. We consider an arbitrary function of the distributed data vector and an arbitrary network with a finite number of nodes with directed or undirected links that have capacity constraints. The setting is best illustrated by an example from [1]. Let $\Theta(X_1, X_2, X_3) = X_1X_2 + X_3$ be the function of interest. The three variables $X_1, X_2,$ and X_3 are generated at sources $s_1, s_2,$ and s_3 respectively. A terminal (or a sink) node t needs to obtain $\Theta(X_1, X_2, X_3)$. We assume that all computations are from the same alphabet $\mathcal{A}$. The computation of Θ can be broken into two parts—first computing X_1X_2, and then adding X_3. These two operations can be done at different nodes in the network in the above order. The sequence of operations to compute Θ is called a *computation schema*, and is represented by a *computation tree* as shown in Fig. 1(b) for the above Θ. We remark that the 'star' tree is a computation tree for any function, i.e., bring the data vector to a single node, compute Θ at that node and transport the result to t. Thus a given function may admit several computation trees. Also, clearly, the knowledge of one or more computations trees is a basic and natural requirement in computing and assuming that they are specified is not a restrictive assumption.

Now consider computing $\Theta(X_1, X_2, X_3)$ in the network shown in Fig. 1(a). Two ways of computing Θ are shown in Figs. 1(c) and 1(d). These are called 'embeddings' (the term will be formally defined in Sec. II). We aim to find the best time-sharing between the various embeddings to achieve the maximum number of computations per use of the network. The approach is to use network flow theory to design a computation and communication scheme that maximizes the rate at which

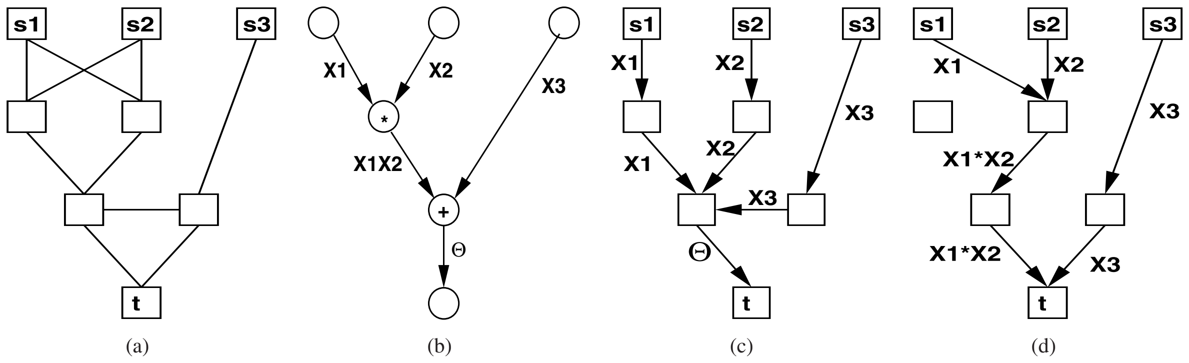

Fig. 1. Computing $\Theta = X_1X_2 + X_3$ over a network. (a) A network to compute $\Theta = X_1X_2 + X_3$. Each link has unit capacity. (b) A schema represented by a computation tree for computing Θ. (c) A possible embedding that computes at Θ at unit rate. (d) An alternative embedding.

Θ is computed. This problem was first described in [1] where we formulated two linear programs (LPs) to determine the optimum flows on each link of the network. These flows are then converted into computing and communicating schedules to compute Θ at the best rate.

Since, a given Θ may allow different computation trees, to find the optimum solution, we need to find the best timesharing between all the embeddings of all such computation trees in the given network. We begin by developing the theory for a single given computation tree. We then describe how the techniques can be used when multiple computation trees are given for Θ. For a given set of computation trees for a function, we obtain the optimum rate at which Θ can be computed by the network.

Organization and contribution: We describe the model in Sec. II. Sec. III presents the main contributions of this paper. For the sake of completeness, we describe the LP from [1], called *Embedding-Edge-LP,* that defines the problem of optimally allocating flows on the embeddings to achieve the maximum computation rate. While this is a natural way to view the problem, we will see that a direct solution to this has an exponential complexity because of the exponential number of possible embeddings. Here, we develop a primal-dual algorithm that is significantly faster than a solution via a *node-arc* LP that was formulated in [1]. Our primal-dual algorithm obtains an approximately optimal (the rate of computation of Θ is within $(1-\epsilon)$ fraction of the optimal rate) solution. As a subroutine for this fast algorithm, we develop an algorithm to find a minimum-weight embedding in a network with weighted links. We believe that the latter is of independent interest. The general technique developed in the Sec. III is applied to other problem situations in Sec. IV where we describe several applications to specific practical problem scenarios—(1) different terminals each wanting a different different function of the data vector, (2) computation with a given desired precision, and (3) networks with energy constraints at nodes.

II. Notation and System Model

The model, and the notation, is similar that in [1]. V is a set of n nodes and E is a set of m undirected edges in the network represented by an undirected, simple, connected graph $\mathcal{N} = (V, E)$. Edge $uv \in E$ is a half duplex link with a total capacity $c(uv) \geq 0$. $S = \{s_1, s_2, \ldots, s_\kappa\} \subset V$ is the set of κ source nodes in $\mathcal{N}$. Source s_i has an infinite sequence of data values $\{X_i(k)\}_{k\geq 0}$ where $X_i(k) \in \mathcal{A}$; $\mathcal{A}$ a finite alphabet. X_i is a representative element of the sequence. Thus the data vector is $X \overset{\triangle}{=} [X_1, \ldots X_\kappa]$. The link capacities are expressed in $|\mathcal{A}|$-ary unit. Without loss of generality, we assume that each source node in the network generates exactly one data sequence. We begin by assuming that there is only one terminal node and consider the case when multiple terminal nodes each of which may want different functions Sec. IV.

The given function $\Theta : \mathcal{A}^\kappa \rightarrow \mathcal{A}$ of X needs to be obtained at t, the terminal, or sink, node at the highest rate possible. $\mathcal{G} = (\Omega, \Gamma)$ is a directed tree that describes the computation schema for Θ; $\mathcal{G}$ is assumed given with Ω as the set of its nodes and Γ as the set of its edges. The elements of Ω are labeled $\mu_1, \mu_2, \ldots, \mu_{|\Omega|}$ with $\mu_1, \mu_2, \ldots, \mu_\kappa$ as the source nodes, $\mu_{|\Omega|}$ as the terminal node that obtains Θ and the others are the computing nodes performing intermediate computations. The nodes in Ω are labeled according to a topological order, i.e., for $i > j$ there is no directed path in $\mathcal{G}$ from μ_i to μ_j. Source nodes have in-degree zero and out-degree one, terminal node has in-degree one and out-degree zero, and other nodes have in-degree greater than one and out-degree exactly one. The elements of Γ are labeled $\theta_1, \theta_2, \ldots, \theta_{|\Gamma|}$ with $\theta_1, \theta_2, \ldots, \theta_\kappa$ being the outgoing edges from $\mu_1, \mu_2, \ldots, \mu_\kappa$ respectively, and $\theta_{|\Gamma|}$ being the incoming edge into $\mu_{|\Omega|}$. For edge $\theta \in \Gamma$, $tail(\theta)$ and $head(\theta)$ are, respectively, the tail and the head nodes of the edge θ. Let $\Phi_\uparrow(\theta)$ and $\Phi_\downarrow(\theta)$ denote, respectively, the predecessors and the successors of θ, i.e., $\Phi_\uparrow(\theta) \overset{\triangle}{=} \{\eta \in \Gamma | head(\eta) = tail(\theta)\}$ and $\Phi_\downarrow(\theta) \overset{\triangle}{=} \{\eta \in \Gamma | tail(\eta) = head(\theta)\}$. The edges are also labeled according to a topological order, i.e., for $i < j$, there is no path from the head node of j to the tail node of i.

We assume that each edge θ of $\mathcal{G}$ represents a distinct function of X that can be computed from the functions corresponding to the edges in $\Phi_\uparrow(\theta)$. Further, each such function takes values from the same alphabet $\mathcal{A}$. This is not unreasonable even when all the computations are over real numbers because computations are performed using a fixed precision.

A sequence of nodes $v_1, v_2, \cdots, v_l$, $l \geq 1$, is called a path if $v_i v_{i+1} \in E$ for $i = 1, 2, \ldots, l-1$. The set of all paths in $\mathcal{N}$ is denoted by $\mathcal{P}$. With abuse of notation, for such a path P, we will say $v_i \in P$ and also $v_i v_{i+1} \in P$. The nodes v_1 and v_l are called respectively the start node and the end node of P, and are denoted as $\mathsf{start}(P)$ and $\mathsf{end}(P)$. We saw in Sec. I that a function with a given computation tree can be computed along any "embedding" of the tree in the network. We now formally define an embedding of a computation tree in the network.

Definition: An embedding is a mapping $B : \Gamma \rightarrow \mathcal{P}$ such that

1) $\mathsf{start}(B(\theta_l)) = s_l$ for $l = 1, 2, \ldots, \kappa$
2) $\mathsf{end}(B(\eta)) = \mathsf{start}(B(\theta))$ if $\eta \in \Phi_\uparrow(\theta)$
3) $\mathsf{end}(B(\theta_{|\Gamma|})) = t$.

Let $\mathcal{B}$ denote the set of embeddings of $\mathcal{G}$ in $\mathcal{N}$. We can now restate our objective—determine the flows on each of the embeddings to maximize the total flow. An edge in the network may carry different functions of the source data in an embedding; thus we define $r_B(e) \triangleq |\{\theta \in \Gamma | e \text{ is a part of } B(\theta)\}|$ to be the number of times an edge $e \in E$ is used in an embedding B. An edge may also be used to carry flows on different embeddings. Therefore in an assignment of flows on different embeddings, i.e., in a particular timesharing scheme, the edge may carry multiple types of data (i.e., different functions of X) of different amounts. Also note that, if $\mathsf{start}(B(\theta_i)) = \mathsf{end}(B(\theta_i))$, i.e., if $B(\theta_i)$ consists of a single node, then in that embedding the data θ_i is generated as well as used (i.e., not forwarded to another node) in that node.

III. A Linear Program and its Efficient Solution

We now present our main contributions. We begin by first describing the basic linear program, the *Embedding-Edge LP* [1], which characterizes our problem.

As discussed in Secs. I and II, the function for a particular sample of the data can be computed over the network using any embedding of the computation tree in the network. For any embedding $B \in \mathcal{B}$, let $x(B)$ denote the average number of function symbols computed using the embedding B per use of the network. The linear program below maximizes $\lambda := \sum_{B \in \mathcal{B}} x(B)$. Recall that $r_B(e)$ represents the number of times the edge e is used in the embedding B.

Embedding-Edge LP: Maximize $\lambda = \sum_{B \in \mathcal{B}} x(B)$ subject to

1. Capacity constraints:

$$\sum_{B \in \mathcal{B}} r_B(e) x(B) \leq c(e), \ \forall e \in E \tag{1}$$

2. Non-negativity constraints:

$$x(B) \geq 0, \ \forall B \tag{2}$$

This LP finds an optimal fractional packing of the embeddings of $\mathcal{G}$ into $\mathcal{N}$.

The cardinality of $\mathcal{B}$ can be exponential in $|V|$. Hence the time complexity of a 'direct solution' of *Embedding-Edge LP* is exponential in the network parameters and is infeasible for large m and we need to exploit the structure of the problem. This issue is similar to that in multi-commodity flow literature where flow conservation at nodes is used to write an equivalent *Node-arc* LP to obtain the flow rates on the links. In [1] we took a similar approach to write down a *Node-Arc LP*, based on a flow conservation principle. It must be noted that the flow conservations here are significantly different than in traditional multi-commodity flow problems and needs a lot more care to develop. The solution to the *Node-Arc LP* is then used to extract a polynomial number of embeddings and assign the flows to each of these embeddings. Here we present a very fast primal-dual based approximate algorithm that can assign the flow rates to the a polynomial number of embeddings to compute the Θ at a rate that is arbitrarily close to the optimum rate, i.e., within $(1-\epsilon)$ fraction of the optimum rate for an $\epsilon > 0$. The complexity of our algorithm is $O\left(\epsilon^{-1} \kappa m (m + n \log n) \log_{1+\epsilon}(m)\right)$, which is faster than any known technique to solve the Node-arc LP.

A. Primal-Dual Algorithm and Min-Cost Embedding

We begin by observing that *Embedding-Edge LP* is a fractional packing problem. For multi-commodity flow packing problems, Garg and Konemann [11] gave a fast primal-dual algorithm to find an approximate solution that is within $(1-\epsilon)$ of the optimal packing. A key requirement of this algorithm is an oracle subroutine that finds the shortest paths between the source-terminal pairs. Based on this technique, we now present a fast, algorithm to obtain an $(1-\epsilon)$-approximate solution to the *Embedding-Edge LP.* We too will use an oracle subroutine but it needs to find the minimum weight embedding in our case. An algorithm for this oracle is also presented here and we believe that this is also of independent interest.

We first write the dual of the *Embedding-Edge LP*. The dual has the variables $L = \{l(e)\}_{e \in E}$ corresponding to the capacity constraints in the primal. The dual LP is given as follows.

Dual of Embedding-Edge LP: Minimize $D(L) = \sum_{e \in E} c(e) l(e)$ subject to

1. Constraints corresponding to each $x(B)$ in primal:

$$\sum_{e \in B} r_B(e) l(e) \geq 1, \ \forall B \tag{3}$$

2. Non-negativity constraints:

$$l(e) \geq 0, \ \forall e \in E \tag{4}$$

We define the weight of an embedding B as

$$w_L(B) = \sum_{e \in B} r_B(e) l(e).$$

Following the method of [11], it can be checked that the dual LP is equivalent to finding $\min_L \frac{D(L)}{\alpha_L}$, where

$$\alpha_l = \min_B w_l(B)$$

is the cost of the minimum cost embedding for L.

For a packing LP of the form

$$\max\left\{a^T x | Ax \leq b, x \geq 0\right\}$$

and its dual LP of the form

$$\min\left\{b^T y | A^T y \geq a, y \geq 0\right\},$$

the shortest path is defined as $\sum_i A(i,j)y(i)/a(j)$ [11]. It can be seen that for the *Embedding-Edge LP*, the 'shortest path' corresponds to the embedding with minimum weight, i.e., $\arg\min_B w_L(B)$. Algorithm 1 below gives the instance of the primal-dual algorithm for the *Embedding-Edge LP*.

Algorithm 1: Algorithm for finding approximately optimal x and λ

```
input : Network graph $\mathcal{N} = (V,E)$, capacities $c(e)$, set
        of source nodes $S$, terminal node $t$, computation
        tree $\mathcal{G} = (\Omega, \Gamma)$, the desired accuracy $\epsilon$
output: Primal solution $\{x(B), B \in \mathcal{B}\}$
1 Initialize $l(e) := \delta/c(e), \forall e \in E, x(B) := 0, \forall B \in \mathcal{B}$ ;
2 while $D(l) < 1$ do
3     $B^*$ := OptimalEmbedding($L$) ;
      // OptimalEmbedding($L$) outputs
      $\arg\min_B w_L(B)$
4     $e^*$ := edge in $B^*$ with smallest $c(e)/r_{B^*}(e)$ ;
5     $x(B^*) := x(B^*) + c(e^*)/r_{B^*}(e^*)$ ;
6     $l(e) := l(e)(1 + \epsilon \frac{c(e^*)/r_{B^*}(e^*)}{c(e)/r_{B^*}(e)})$, $\forall e \in B^*$ ;
7 end
8 $x(B) := x(B)/\log_{1+\epsilon} \frac{1+\epsilon}{\delta}, \forall B$ ;
```

We now describe, and then provide below, the subroutine *OptimalEmbedding(L)* that obtains a minimum weight embedding of $\mathcal{G}$ on $\mathcal{N}$ with a given length/cost function L. For each edge θ_i, starting from θ_1, the algorithm finds a way to compute θ_i at each network node at the minimum cost possible. It keeps track of that minimum cost and also the 'predecessor' node from where it receives θ_i. If θ_i is computed at that node itself then the predecessor node is itself. This is done for each θ_i by a technique similar to Dijkstra's shortest path algorithm. Computing θ_i for $i \in \{1, 2, \ldots, \kappa\}$ at the minimum cost at a node u is equivalent to finding the shortest path to u from s_i. We do this by using Dijkstra's algorithm. For any other i, the node u can either compute θ_i from $\Phi_\uparrow(\theta_i)$ or receive it from one of its neighbors. To take this into account, unlike Dijkstra's algorithm, we initialize the cost of computing θ_i with the cost of computing $\Phi_\uparrow(\theta_i)$ at the same node. With this initialization, the same principle of greedy node selection and cost update as in Dijkstra's algorithm is used to find the optimal way of obtaining θ_i at all the nodes. Finally, the optimal embedding is obtained by backtracking the predecessors. Starting from t, we backtrack using predecessors from which $\theta_{|\Gamma|}$ is obtained, till we hit a node whose predecessor is itself. This node is the start node of $B(\theta_{|\Gamma|})$ and the end node of $B(\eta)$ for all $\eta \in \Phi_\uparrow(\theta_{|\Gamma|})$. The complete embedding is obtained by continuing this process for each θ_i in the reverse topological order. This is described in Procedure 2.

Procedure OptimalEmbedding(L)

```
input : Network graph $\mathcal{N} = (V,E)$, Length function $L$,
        set of source nodes $S$, terminal node $t$,
        computation tree $\mathcal{G} = (\Omega, \Gamma)$.
output: Embedding $B^*$ with minimum weight under $L$
1  for $i = 1$ to $|\Gamma|$ do
2      if $i \in \{1, 2, \ldots, \kappa\}$ then
3          $\omega_u(\theta_i) := \infty, \forall u \in V - \{s_i\}$ ;
4          $\omega_{s_i}(\theta_i) := 0$ and $\sigma_{s_i}(\theta_i) := s_i$ ;
5      end
6      else
7          $\omega_u(\theta_i) := \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta), \forall u \in V$ ;
8          $\sigma_u(\theta_i) := u, \forall u \in V$ ;
9      end
10     $\Psi := \emptyset$; $\bar{\Psi} := V$ ;
11     while $|\Psi| < n$ do
12         $v := \arg\min_{u \in \bar{\Psi}} \omega_u(\theta_i)$ ;
13         $\Psi := \Psi \cup \{v\}$ ;
14         $\bar{\Psi} := \Psi - \{v\}$ ;
15         foreach $u \in N(v)$ do
16             if $\omega_v(\theta_i) + l(uv) < \omega_u(\theta_i)$ then
17                 $\omega_u(\theta_i) := \omega_v(\theta_i) + l(uv)$ ;
18                 $\sigma_u(\theta_i) := v$ ;
19             end
20         end
21     end
22 end
23 $B^*(\theta_{|\Gamma|}) := t$ ;
24 for $i = |\Gamma|$ to $1$ do
25     $u$ := end($B^*(\theta_i)$) ;
26     while $\sigma_u(\theta_i) \neq u$ do
27         Prefix $\sigma_u(\theta_i)$ to $B^*(i)$ ;
28         $u := \sigma_u(\theta_i)$ ;
29     end
30     $B(\eta) := u \; \forall \eta \in \Phi_\uparrow(\theta_i)$ ;
31 end
```

Correctness of *OptimalEmbedding(L)*: To prove the correctness, it is sufficient to show that, during each phase i, the algorithm computes optimal values for $\omega_u(\theta_i)$ and $\sigma_u(\theta_i)$, for each node u in $\mathcal{N}$. We prove this by induction on the pair $(i, |\Psi|)$ according to the lexicographic ordering. For $i \in \{1, \ldots, \kappa\}$ and for all $|\Psi|$, this follows from the correctness of Dijkstra's algorithm. Now, assuming the optimality of $\omega_u(\theta_i)$ and $\sigma_u(\theta_i)$ till all iterations before $(i, |\Psi|)$, we prove the statement for $(i, |\Psi|)$. Suppose v is the element added to Ψ in the current iteration. We consider two cases:

Case 1: $\Psi = \{v\}$: The cost of computing (and not receiving from another node) θ_i at any node u is $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$. The algorithm chooses v which has the minimum $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$ among all nodes $u \in V$ and assigns $\omega_v(\theta_i) = \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_v(\eta)$ and $\sigma_v(\theta_i) = v$. If these are not optimal, then it must be more efficient for v

to receive θ_i which is computed at some other node u. But that implies $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta) < \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_v(\eta)$, which is a contradiction to the choice of v.
Case 2: $\{v\} \subsetneq \Psi$: Suppose there is a more efficient way of receiving θ_i at v than from the node selected as $\sigma_v(\theta_i)$ and that is to compute θ_i at a node u and receive it along a path $P_{u,v}$. Let the corresponding cost be $\omega'_v(\theta_i)$. First, if $u \in \Psi'$, then the present cost $\left(\leq \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)\right)$ at u is less than the present value of $\omega_v(\theta_i)$, which is a contradiction to the choice of v. Thus $u \in \Psi$. Let u' be the last node in $P_{u,v}$ from Ψ, and v' be the first node in $P_{u,v}$ from Ψ'. Then $\omega'_v(\theta_i) \geq \omega_{u'}(\theta_i) + l(u'v') \geq \omega_{v'}(\theta_i) \geq \omega_v(\theta_i)$ — a contradiction. Here the first inequality follows since $u' \in \Psi$. The second inequality follows from the update rule followed during the inclusion of u' in Ψ. The last inequality follows from the choice of v.

Complexity of *OptimalEmbedding(L)* and the Primal-Dual algorithm: Let us consider the first *for* loop in *OptimalEmbedding(L)*. Each iteration of this loop is the same as Dijkstra's algorithm except for the initialization. Thus, the for loop, excluding the initialization step, can be run in $O(m + n \log n)$ time using Fibonacci heap implementation. The initialization step requires $O(n|\Phi_\uparrow(\theta_i)|)$ time for each iteration. The second *for* loop has $O(n\kappa)$ complexity. So the overall algorithm takes $O(\kappa(m + n \log n))$ time.

The number of iterations in the primal-dual algorithm is of the order $O(\epsilon^{-1} m \log_{1+\epsilon}(m))$. Thus the overall complexity of the algorithm is $O\left(\epsilon^{-1} \kappa m(m + n \log n) \log_{1+\epsilon}(m)\right)$. This is faster than known algorithms that solve the *Node-arc* LP; the scheme of [12] has a time complexity of $O\left((m\kappa)^{2.5}\right)$ while that of [13] has a time complexity of $O\left((m\kappa)^{3.5}\right)$.

B. Multiple trees for Θ

We now use the technique described above for the more general case when there are multiple computation trees for a given function Θ. For example, the 'sum' function $f(X_1, X_2, X_3) = X_1 + X_2 + X_3$ may be computed by any of the computation sequences $\big((X_1 + X_2) + X_3\big)$, $\big(X_1 + (X_2 + X_3)\big)$, or $\big(X_2 + (X_1 + X_3)\big)$. In general, suppose multiple computation trees $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_\nu$ are given for computing the same function. Let $\mathcal{B}_i$ denote the set of all embeddings of $\mathcal{G}_i$ for $i = 1, 2, \ldots, \nu$. Let $\mathcal{B} = \cup_i \mathcal{B}_i$ denote the set of all embeddings. Under this definition of $\mathcal{B}$, the *Embedding-Edge LP* for this problem is the same as that for a single tree. The new *OptimalEmbedding(L)* algorithm finds an optimal embedding for each $\mathcal{G}_i$ and chooses the one with minimum weight as the optimal embedding in $\mathcal{B}$. This can be used in the same primal-dual algorithm to find an $(1 - \epsilon)$-approximate solution.

Some edges of different trees may represent an identical function of the sources. For example, for the function $X_1 + X_2 + X_3 + X_4$, an edge corresponding to the function $X_1 + X_2$ is present in each of the trees corresponding to $\Big(\big((X_1 + X_2) + X_3\big) + X_4\Big)$, $\big((X_1 + X_2) + (X_3 + X_4)\big)$, and $\Big(\big((X_1 + X_2) + X_4\big) + X_3\Big)$. For this reason, *OptimalEmbedding(L)* algorithm can be made more efficient by running iterations for each function rather than each edge. The initialization of $\omega_u(\theta)$ changes correspondingly, to take into account all possible ways of computing that function. Rest of the algorithm remains the same.

The particular function $\Theta(X_1, X_2, \ldots, X_\kappa) = X_1 + X_2 + \ldots + X_\kappa$ is of special theoretical and practical interest. There are $\kappa!$ sequences of additions of data and corresponding trees to obtain this function. With the above modification, our *OptimalEmbedding(L)* algorithm has complexity exponential in κ and linear in m. As a result, our primal-dual algorithm gives an $(1 - \epsilon)$-approximate solution in exponential complexity in κ and quadratic in m. This is not unexpected since the problem is equivalent to the much investigated multicast problem which is well-known to be NP-hard in κ. However, note that our technique suggests a suboptimal technique of considering only a subset of all possible computation trees, which would result in sub-optimal performance. The tradeoff of restricting embeddings and reducing the overall complexity with suboptimality of the solution, though beyond the scope of this paper, is an interesting avenue for further study.

IV. Applications

We now describe how to apply the techniques that we developed in the previous section to three practical problems. This is, of course, a sample of the applications of our techniques.

1. Multiple functions and multiple terminals: Suppose that the network has multiple terminals $t_1, t_2, \ldots, t_\gamma$ wanting, respectively, functions $\Theta_1(X^{(1)}), \Theta_2(X^{(2)}), \ldots, \Theta_\gamma(X^{(\gamma)})$. Here $X^{(i)}$ is the data generated by a set of sources $S^{(i)}$. The sets $S^{(i)}; i = 1, 2, \ldots, \gamma$ are assumed to be pairwise disjoint. For each function Θ_i, a computation tree $\mathcal{G}_i$ is given. Let us consider the problem of communicating the functions to the respective terminals at rates $\lambda_1, \lambda_2, \ldots, \lambda_\gamma$. The problem is to determine the achievable rate region which is defined as the set of $\mathbf{r} = (\lambda_1, \lambda_2, \ldots, \lambda_\gamma)$ for which a protocol exists for transmission of the functions at these rates. This region can be approximately found by solving either of the following problems.

(i) For any given non-negative weights $\alpha_1, \alpha_2, \ldots, \alpha_\gamma$, what is the maximum achievable weighted sum-rate $\sum_{i=1}^{\gamma} \alpha_i \lambda_i$? For this problem, we consider embeddings of the computation trees $\mathcal{G}_i$ into the network for each terminal t_i. Let $\mathcal{B}_i$ denote the set of all embeddings of $\mathcal{G}_i$. Then the *Embedding-Edge LP* for this problem is to maximize $\sum_{i=1}^{\gamma} \alpha_i \sum_{B \in \mathcal{B}_i} x(B)$. The constraints are the same as before with $\mathcal{B}$ defined by $\mathcal{B} = \cup_i \mathcal{B}_i$. The weight of an embedding $B \in \mathcal{B}$ under a weight function L is defined as $\alpha_i w_L(B)$ if $B \in \mathcal{B}_i$. The new *OptimalEmbedding(L)* algorithm finds an optimal embedding for each $\mathcal{G}_i$ and chooses the one with minimum weight. This can be used in the same primal-dual algorithm to find an $(1 - \epsilon)$-approximate solution.

(ii) For any non-negative demands $\alpha_1, \alpha_2, \ldots, \alpha_\gamma$, what is the maximum λ for which the rates $\lambda\alpha_1, \lambda\alpha_2, \ldots, \lambda\alpha_\gamma$ are concurrently achievable?

Here, we define an embedding to be a tuple $B = (B_1, B_2, \ldots, B_\gamma)$, where $B_i \in \mathcal{B}_i$ is an embedding of the computation tree $\mathcal{G}_i$. The *Embedding-Edge LP* for this problem is the same as that for the single terminal problem with $r_B(e)$ defined as $r_B(e) = \sum_{i=1}^{\gamma} \alpha_i |\{\theta \in \Gamma_i | e \text{ is a part of } B_i(\theta)\}|$ and $\mathcal{B} = \mathcal{B}_1 \times \mathcal{B}_2 \times \ldots \times \mathcal{B}_\gamma$. The weight of an embedding B under a weight function L is defined as $\sum_{i=1}^{\gamma} \alpha_i w_L(B_i)$. The new *OptimalEmbedding(L)* algorithm finds an optimal embedding B by separately finding optimal embeddings B_i for each $\mathcal{G}_i$. This can be used in the same primal-dual algorithm to find an $(1-\epsilon)$-approximate solution.

2. Computing Θ with a specified precision: In practice, the source data may be real-valued, and communicating such a data requires infinite capacity. In such applications, it is common to require a quantized value of the function at the terminal with a desired precision. This may, in turn, be achieved by quantizing various data types with pre-decided precisions and thus different data type may require different number of bits to represent them. Suppose the data type denoted by θ is represented using $b(\theta)$ bits. Then the *Embedding-Edge LP* and its dual for this problem are the same as before except that the definition of $r_B(e)$ is changed to $r_B(e) = \sum_{\theta \in \Gamma : e \text{ is a part of } B(\theta)} b(\theta)$. In the *OptimalEmbedding(L)* algorithm, $l(uv)$ is replaced by $l(uv)b(\theta_i)$ inside the *foreach* loop.

3. Energy limited sensors: Suppose, instead of capacity constraints on the links, each node $u \in V$ has a total energy $E(u)$. Each transmission and reception of θ require the energy $E_{T,\theta}$ and $E_{R,\theta}$ respectively. Generation of one symbol of θ or computation of one symbol of θ from $\Phi_\uparrow(\theta)$ requires the energy $E_{C,\theta}$. For power-limited but band-unlimited (or practically time-unlimited) links, or for a fixed modulation/coding scheme in use, it is reasonable to assume that the energy consumed is proportional to the number of transmissions. The objective now is to compute the function at the terminal maximum number of times with the given total node energy at each node.

For an embedding B, if $B(\theta) = v_1, v_2, \cdots, v_l$, then $tr(B(\theta)) = \{v_1, v_2, \cdots, v_{l-1}\}$ denotes the transmitting nodes, and $rx(B(\theta)) = \{v_2, v_3, \cdots, v_l\}$ denotes the receiving nodes of θ. If $l = 1$, then $tr(B(\theta)) = rx(B(\theta)) = \emptyset$. For B, the energy load on the node u is given by

$$E_B(u) = \sum_{\theta : \text{start}(B(\theta)) = u} E_{C,\theta} + \sum_{\theta : u \in tx(B(\theta))} E_{T,\theta} + \sum_{\theta : u \in rx(B(\theta))} E_{R,\theta}.$$

The capacity constraint in the *Embedding-Edge LP* is replaced by the energy constraint on the nodes

$$\sum_{B \in \mathcal{B}} x(B) E_B(u) \leq E(u) \;\forall u \in V,$$

where an empty sum is defined to be 0. The dual of the *Embedding-Edge LP* is: Minimize $D(L) = \sum_{u \in V} E(u) l(u)$ subject to

1. Constraints corresponding to each $x(B)$ in primal:

$$\sum_{u \in B} E_B(u) l(u) \geq 1, \;\forall B \qquad (5)$$

2. Non-negativity constraints:

$$l(u) \geq 0, \;\forall u \in V. \qquad (6)$$

The weight or cost of an embedding can be defined as

$$w_L(B) = \sum_{u \in B} E_B(u) l(u).$$

The *OptimalEmbedding(L)* is modified in the weight initialization and weight update. The weight initialization is done as $\omega_{s_i}(\theta_i) := E_{C,\theta_i}$ for source data and $\omega_u(\theta_i) := E_{C,\theta_i} + \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$ for other data. The weight update at u is now done as $\omega_u(\theta_i) := \omega_v(\theta_i) + E_{T,\theta_i} + E_{R,\theta_i}$ if $\omega_v(\theta_i) + E_{T,\theta_i} + E_{R,\theta_i} < \omega_u(\theta_i)$. After suitable modification, the primal-dual algorithm with the modified *OptimalEmbedding(L)* algorithm finds an $(1-\epsilon)$-approximate solution.

V. Discussion and conclusion

In this paper we approached the problem of optimal in-network function computation from a more practical perspective. For reasonable sized networks, we use the theory of network flows to develop a flow allocation based algorithm to compute the desired functions optimally, with a suitably defined optimality criterion. We presented a fast algorithm to obtain an approximately optimal flow allocation which can be used to devise a communication and computing schedule. Our framework can be easily used to develop an LP to find an optimum static scheduling scheme for wireless networks with interference-induced link scheduling constraints. We have omitted that discussion due to space constraints.

References

[1] V. Shah, B. K. Dey, and D. Manjunath, "Network flows for functions," *Submitted to ISIT 2011; available at* `http://www.ee.iitb.ac.in/~bikash/isit2011.pdf`.

[2] T. S. Han and K. Kobayashi, "A dichotomy of functions $f(x, y)$ of correlated sources (x, y)," *IEEE Trans. Inform. Theory*, vol. 33, no. 1, pp. 69–86, 1987.

[3] A. Orlitsky and J. R. Roche, "Coding for computing," *IEEE Trans. Inform. Theory*, vol. 47, no. 3, pp. 903–917, 2001.

[4] E. Kushilevitz and N. Nisan, *Communication Complexity*, Cambridge University Press, 1997.

[5] G. Liang and N. H. Vaidya, "Error-free multi-valued consensus with byzantine failures," *CoRR*, vol. abs/1101.3520, 2011.

[6] R. G. Gallager, "Finding parity in simple broadcast networks," *IEEE Trans. on Info. Theory*, vol. 34, pp. 176–180, 1988.

[7] A. Giridhar and P. R. Kumar, "Computing and communicating functions over sensor networks," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 4, pp. 755–764, April 2005.

[8] S. Kamath and D. Manjunath, "On distributed function computation in structure-free random networks," in *Proc. of IEEE ISIT*, Toronto, Canada, July 2008.

[9] Brijesh Kumar Rai and Bikash Kumar Dey, "Sum-networks: system of polynomial equations, reversibility, insufficiency of linear network coding, unachievability of coding capacity," *available at http://arxiv.org/abs/0906.0695*.

[10] R. Appuswamy, M. Franceschetti, N. Karamchandani, and K. Zeger, "Network coding for computing : Cut-set bounds," *available at http://arxiv.org/abs/0912.2820*, 2010.

[11] N. Garg and J. Konemann, "Faster and simpler algorithms for multicommodity flow and other fractional packing problems," in *Proc. of 39th Ann. Symp. on FOCS*, Nov. 1998, pp. 300–309.

[12] P. Vaidya, "Speeding-up linear programming using fast matrix multiplication," in *Proc. of the 30th Ann. Symp. on FOCS*, 1989.

[13] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proc. of the 16th Annual ACM STOC*, 1984, pp. 302–311.