SOAR: Minimizing Network Utilization with Bounded In-network Computing

Raz Segal, Chen Avin, Gabriel Scalosub
School of Electrical and Computer Engineering
Ben-Gurion University of the Negev, Israel

ABSTRACT

In-network computing via smart networking devices is a recent trend for modern datacenter networks. State-of-the-art switches with near line rate computing and aggregation capabilities are developed to enable, e.g., acceleration and better utilization for modern applications like big data analytics, and large scale distributed and federated machine learning. We formulate and study the problem of activating a limited number of in-network computing devices within a network, aiming at reducing the overall network utilization for a given workload. Such limitations on the number of in-network computing elements per workload arise, e.g., in incremental upgrades of network infrastructure, and are also due to requiring specialized middleboxes, or FPGAs, that should support heterogeneous workloads, and multiple tenants.

We present an optimal and efficient algorithm for placing such devices in tree networks with arbitrary link rates, and further evaluate our proposed solution in various scenarios and for various tasks. Our results show that having merely a small fraction of network devices support in-network aggregation can lead to a significant reduction in network utilization. Furthermore, we show that various intuitive strategies for performing such placements exhibit significantly inferior performance compared to our solution, for varying workloads, tasks, and link rates.

ACM Reference Format:
Raz Segal, Chen Avin, Gabriel Scalosub. 2021. SOAR: Minimizing Network Utilization with Bounded In-network Computing . In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485983.3494853

1 INTRODUCTION

Datacenter networks and their distributed data processing capabilities are the driving force behind leading applications and services, including search engines, content distribution, social networks and eCommerce. Recent work has shown that for many of the tasks performed by such applications, the network (and not server computation) is the actual bottleneck hindering the ability to optimize computation efficiency and performance [12, 34, 52]. Such bottlenecks occur, e.g., in distributed and federated machine learning (e.g., *AllReduce*), and in solutions employing the MapReduce methodology for big data tasks, and more generally in scenarios giving rise to the *incast* problem [5, 55].

In order to tackle these deficiencies, recent research has been pushing the concept of *in-network computing* [41, 43], which suggests offloading a considerable portion of the computation onto "smart" networking elements, thus relieving end-hosts and servers from some of the computational tasks, resulting in improved efficiency and performance. In proposing this paradigm, attempts were made to characterize the types of computation that could potentially benefit from such an approach [13]. Indeed, recent works show that modern switches can perform local computation on packets, like reduce operations, even at line rate [19, 21]. Such *computing switches* can be implemented, for example, using SDN and programmable network elements (e.g., using P4) [9], and have been shown to significantly improve network utilization, which in turn improves overall application performance, and resource usage efficiency [19, 21]. It should be noted that the question of whether such offloading approaches are beneficial or advised is not without controversies [36]. However, data aggregation, as performed in, e.g., big-data tasks based on MapReduce, and distributed ML, which are the main use cases considered in our work, are well within consensus, especially when implemented using programmable switches with co-located accelerators (such as FPGAs).

Bearing these potential benefits in mind, one should note that employing in-network computing comes at a cost (in the form of, e.g., hardware or availability limitations), and such capabilities might not be ubiquitous throughout the network. For example, this could be the case in an incremental upgrade of the network, where some (but not all) legacy switches are replaced by more capable network elements. In addition, in many cases, such in-network computing require specialized middleboxes, or FPGAs, which might call for independent, possibly partial, deployment. Lastly, even if such in-network computing capabilities are indeed available throughout the network, the available resources required to support the various workloads requiring such computation might not be sufficient for servicing all such workloads. In such a case, one would need to allocate in-network computing resources sparingly to the various workloads, to optimize overall system performance. We therefore focus our attention on in-network computation tasks, while using a *limited* number of in-network processing devices.

In particular, we consider the task of *data aggregation* as it occurs in, e.g., MapReduce frameworks, or distributed machine learning using a parameter server. We study such in-network computing paradigms in tree-based topologies where given a tree network of switches, each connected to some number of servers (e.g., as Top-of-Rack switches), our goal is to perform data aggregation by means of a *Reduce* operation; We wish to send the aggregated data

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9098-9/21/12…$15.00
https://doi.org/10.1145/3485983.3494853

from all the servers in the network, towards a special *destination* server d. We note that such tree-based topologies are becoming increasingly popular for distributed machine-learning use cases, implementing, e.g., AllReduce operations [19, 22, 42].

A simple example of our problem is depicted in Fig. 1, where the destination server d is connected via a tree to six servers. Initially, each server i holds a value x_i and d needs to compute a function $f(x_1, x_2, \ldots, x_6)$ over all the values available at the servers. To perform this task more efficiently, we have at our disposal a limited *budget* of k aggregation switches, which should be deployed (or activated) in some k locations in the tree network. While several different metrics of interest could be considered, in the current work we focus on optimizing the *utilization complexity*, where one strives to minimize the *total transmission time* throughout the network while performing the Reduce operation. This is equivalent to minimizing the average transmission time over all links. When link rates are the same across the network (e.g., rate 1), the utilization complexity is proportional (or even identical) to the message complexity [40], the total number of messages sent during the operation. In this work we consider the more general case of having arbitrary rates at the links. The utilization complexity therefore serves as a generalization of message complexity, and can be considered as a basic metric for the performance of network algorithms, where we apply it to studying the efficiency of the Reduce operation. We note that for a given network capacity induced by the link rates, the ability to maintain a low utilization complexity is expected to allow supporting more workloads.

The benefit of having an aggregation switch at some location is that such a switch can perform local aggregation of messages, i.e., aggregating multiple incoming messages onto a *single* outgoing message. Hence, the judicious allocation of these aggregation switches can assist in significantly reducing the utilization complexity. It should be noted that the utilization complexity is closely correlated with the actual *bandwidth consumption* (in Bytes) of the system, while performing a Reduce operation (as we further demonstrate in Sec. 5).

To better illustrate the notion of utilization complexity, assume, for example, that server d needs to compute an aggregate function f (e.g., sum) of the x_i's in Fig. 1. Two extreme in-network allocations are (i) the *all-red* solution, where no switches serve as aggregation switches, (i.e., $k = 0$), and (ii) the *all-blue* solution, where all switches are aggregation switches, thus requiring the allocation of $k = 5$ aggregation switches. If we assume for simplicity that all link rates are 1, the all-red solution translates to having a utilization complexity of 14, as there is an overall of 14 messages traversing the network, where the all-blue solution will require merely 5 – the number of edges in the tree.

As it turns out, for non-extremal cases of k, finding the optimal placement of the aggregation switches is not a trivial task, even for trees, which is the case being studied in this work. This follows from the fact that the optimal placement of the aggregation switches depends both on the (possibly complex) tree topology and links rates, as well as on the (possibly complex) load distribution at the servers. Moreover, multiple aggregation switches on the unique path from a server to the tree root introduce dependencies between the switches, which render standard approaches, like greedy, or divide and conquer, inapplicable.

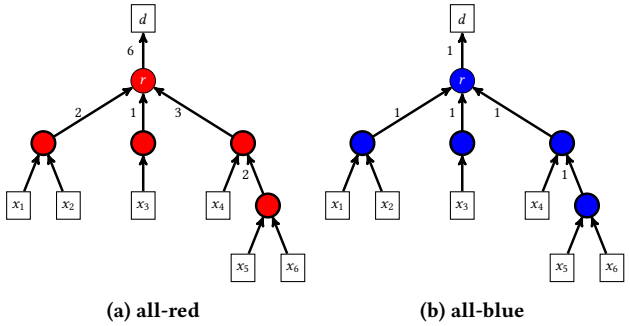


Figure 1: The number of messages on each edge in *all-red* and *all-blue* aggregation trees. Destination computes a function f on $x_1, \ldots, x_6$.

We believe that our problem setup could also be used, for example, by cloud providers that can offer such a service as part of their Network-as-a-Service (NaaS) offerings, where each client can choose its required amount of aggregation switches based on the performance it needs.

We note that our work focuses primarily on reducing the bandwidth footprint, thus maximizing the effective utilization of the networking resources. More recently, focus has also been given to highlighting networking bottlenecks that are due to transport-level deficiencies, which hinder exploiting the full potential of distributed applications such as big data tasks and ML [62]. Our proposed approach can be applied alongside any solutions being thus developed for other layers of the networking stack.

1.1 Our Contribution

We formulate the *Bounded In-network Computing* (BIC) problem, aiming at minimizing the utilization complexity, and present an optimal and time-efficient algorithm for solving the problem on tree networks with arbitrary, heterogeneous, link rates. Such topologies are common in datacenter networks, e.g., fat-tree topologies [4]. Our algorithm uses dynamic programming with a non-trivial parameterized potential function.

While our mathematical formulation is for a single workload (or tenant), we extend it to support multiple workloads that arrive in an *online* manner, each requiring the allocation of (some) in-network aggregation switches. In such a scenario,n each switch has a limited capacity of workloads it can support. We discuss and present various properties of our resulting solution, and evaluate its performance for various server load distribution, network sizes, and network topologies. In our study, we further consider two main *use cases*: (i) MapReduce (using word-count as an illustration), and (ii) gradient aggregation in distributed machine learning using a parameter server. We further show the benefits of using our algorithm when compared with several natural allocation strategies. Our results indicate that a small fraction of aggregation switches can already significantly diminish the utilization complexity of data aggregation tasks.

While we use an abstract mathematical model for scatter-gather type applications, we believe the model, and our algorithmic approach, alongside the structural properties it uncovers, may well

be suited for further studying other objectives like minimizing the load on bottleneck links, or minimizing the latency of completing the data-transfers of a workload.

The rest of the paper is structured as follows. In Sec. 2 we introduce our formal system model. Sec. 3 provides a motivating example highlighting various aspects of the *BIC* problem. Sec. 4 presents an overview of our optimal algorithm SOAR and the main theoretical results. We evaluate our algorithm experimentally in Sec. 5. The formal algorithms and analysis are presented in Section 6. We conclude the paper with related work and discussion in Secs. 7 and 8, respectively. We note that due to space constraints, some of the proofs have been omitted, and can be found in [45].

2 PRELIMINARIES & SYSTEM MODEL

We consider a system comprising a set of n switches $\mathcal{S}$, a set of servers (workers) $\mathcal{W}$, and a special destination server $d \notin \mathcal{W}$. We assume there exists a pre-specified *root* switch $r \in \mathcal{S}$, and a *weighted tree network* $T = (V, E, \omega)$, where $V = \mathcal{S} \cup \{d\}$ and $E = E' \cup \{(r, d)\}$ for some $E' \subseteq \mathcal{S}^2$ forming a tree over the set of switches $\mathcal{S}$. Let $\omega : E \mapsto \mathbb{R}^+$ be the rate function of the links (in messages per second). For $e \in E$ let $\rho(e) = \frac{1}{\omega(e)}$. The tree T thus consists of the underlying network topology connecting the switches, and connecting the root r to the destination d.

We further assume that all links in E are directed towards d. Let $\tau(u, v)$ denote the unique directed path from u to v, if such exist. In particular, every switch $s \in \mathcal{S}$ has a unique *parent* switch $p(s) \in \mathcal{S}$ defined as the neighbor of s on the unique path from s to the d, $\tau(s, d)$. In such a case we say s is a *child* of $p(s)$, and we let $C(s)$ denote the number of children of switch s. When v is an ancestor of u let $\rho(v, u) = \sum_{e \in \tau(v,u)} \rho(e)$, and for convenience we let $\rho(s) = \rho((s, p(s)))$.

We use $D(s)$ to denote the distance between switch s and the root r, and let $h(T) = \max_s D(s)$ denote the *height* of the tree T.

We assume each server $w \in \mathcal{W}$ is connected to a single switch $s(w) \in \mathcal{S}$, and let $L : \mathcal{S} \mapsto \mathbb{N}$ be the function matching each switch s with the number of servers connected to s. We refer to L as the *network load*. Each server w produces a single message, x_w, which is forwarded to $s(w)$, where we assume every message has size at most M, for some (large enough) constant M. The destination server d needs to compute some function $f(\overline{x})$. Each switch s can be of one of two types, or operates at one of two modes:

(i) an *aggregating* switch (blue), which can aggregate messages arriving from its children (each of size at most M), to a *single* message (also of size at most M) and forwards it to its parent switch $p(s)$,[1] or
(ii) a *non-aggregating* switch (red), which cannot aggregate messages, and simply forwards each message arriving from any of its children to its parent switch $p(s)$.

We denote by $\Lambda \subseteq \mathcal{S}$ the set of switches that are *available* to serve as aggregation switches.

Our assumption on the aggregation capabilities of aggregating switches is satisfied by systems computing, e.g., separable functions [37]; A separable function of two independent values can be expressed as the product-operator of the values of two individual functions, each one applied to a distinct operand value. In particular, this holds true for aggregation functions computing, e.g., the count, sum, or max/min of the values contained in the messages being sent by the servers [20]. We leave for future work the study of more complex functions.

[1] We assume M is large enough to hold the value of the function f being computed at every node.

Algorithm 1 Reduce (T, L, U)

Require: tree T, network load L, set of blue nodes U
Ensure: aggregate information at destination d
1: for each node v in T do:
2: **while** not received all messages from all children **do**
3: process incoming message (by switch type: B, R)
4: if needed send message to $p(v)$ (by switch type: B, R)

In what follows we will be referring to aggregating switches as *blue* nodes in T, and to non-aggregating switches as *red* nodes in T. We denote by a non-negative integer k our *budget*, which serves as an upper bound on the number of blue nodes allowed in T. We will usually refer to $U \subseteq \Lambda$ as the set of blue nodes in T and require that $|U| \leq k$.

Given a weighted tree network $T = (V, E, \omega)$ with a network load $L : \mathcal{S} \mapsto \mathbb{N}$, and a set of blue nodes $U \subseteq \Lambda$, we consider a simple Reduce operation on T as detailed in Algorithm 1. Every switch in the tree processes all messages received from its children and forwards message(s) to its parent. Every blue node (i.e., a node in U) is an aggregation switch and all other switches (i.e., nodes not in U) are non-aggregation switches. The operation ends when the *destination* receives the overall (possibly aggregated) information from all the nodes that have a strictly positive load.

For every link $e = (s, p(s))$ in E, we then define the *link message cost* $\text{msg}_e(T, L, U)$ as the number of messages traversing link e, given the Reduce operation on T, L, and U. We further define the *network utilization cost* (or the utilization complexity) as the total transition time associated with the Reduce operation on T, U and L to be

$$\phi(T, L, U) = \sum_{e \in E} \text{msg}_e(T, L, U) \cdot \rho(e). \tag{1}$$

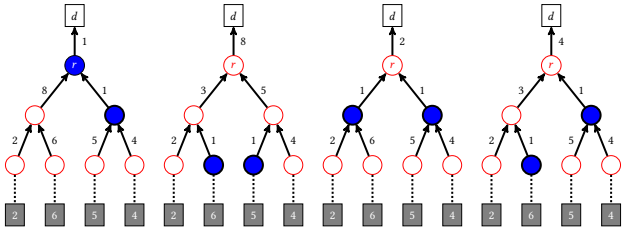
The network utilization complexity measures the total (or equivalently, the average) transmission time of all links in performing the Reduce operation.

In this paper we study the *Bounded In-network Computing* (*BIC*) allocation problem which tries to minimize the network utilization cost. We refer to this problem as the ϕ-*BIC* problem, which is formally defined as follows:

Definition 2.1 (ϕ-*BIC*)**.** Given a weighted tree network $T = (V, E, \omega)$, a network load $L : \mathcal{S} \mapsto \mathbb{N}$, a set of available switches Λ, and a budget k, the ϕ-*BIC* problem is finding a set of switches $U \subseteq \Lambda$ of size at most k that minimizes the utilization cost $\phi(T, L, U)$. Formally,

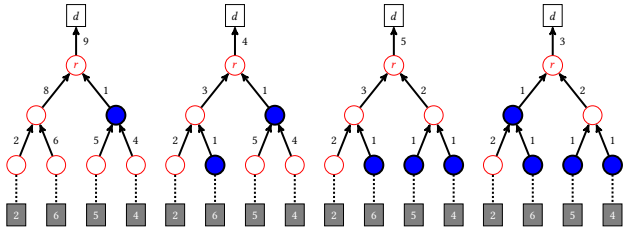
$$\phi\text{-}BIC(T, L, \Lambda, k) = \min_{U \subseteq \Lambda, |U| = k} \phi(T, L, U). \tag{2}$$

Clearly, one can use a brute-force approach, considering all possible $\Theta(n^k)$ subsets of size k. Although this approach may work well for a small constant k, such enumeration would result in exorbitant running time for arbitrary values of k. In what follows we describe and discuss our *efficient* solution, SOAR, to the ϕ-*BIC* problem.



(a) Top (b) Max (c) Level (d) SOAR

Figure 2: Example of solutions produced by 4 allocation algorithms for solving ϕ-*BIC*, with constant rates of 1, $\Lambda = \mathcal{S}$ and $k = 2$ aggregation switches allowed (blue nodes). Switches are marked by circles, the servers connected to each of the leaf switches are depicted by a gray square noting the load of the switch, and the destination is marked by a white square. In each solution, each link in T is marked by its utilization value. Subfigures (2a), (2b), (2c), **and** (2d) **show the solutions produced by Top, Max, Level, and SOAR, respectively, obtaining network utilization costs of 27, 24, 21, and 20, respectively.**



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$ (d) $k = 4$

Figure 3: Example of optimal solutions (produced by SOAR) for distinct bounds on the number of allowed blue switches $k = 1, 2, 3, 4$, with network utilization costs of 35, 20, 15, 11, respectively. The settings are the same as in Fig. 2. The solutions for $k = 2$ (Subfigure (3b)**) and for $k = 3$ (Subfigure** (3c)**) are unique. These two cases serve as an example for the fact that the optimal sets of blue nodes for increasing values of k are not necessarily monotone.**

3 MOTIVATING EXAMPLE

We now turn to consider a motivating example highlighting the fact that simple, yet reasonable, approaches might fall short of finding an optimal solution to the ϕ-*BIC* problem. Specifically, we consider the following three allocation strategies for determining the set of blue nodes: (i) The Top strategy, which picks the set of k blue nodes as the set closest to the root. This approach targets reducing the number of messages transmitted in the topmost part of the network, and is motivated by the fact that failing to aggregate messages close to the root may lead to a large number of messages being forwarded from the root to the destination. (ii) The Max strategy, which picks the set of blue nodes as the k switches with the largest load. (iii) The Level strategy, defined for complete binary trees, which aims at partitioning the network into subtrees of similar size, where all the messages within a subtree are aggregated. This is done by picking a whole level in the complete binary tree as the set of blue nodes.

We consider a tree network with $n = 7$ switches, inducing a complete binary tree topology on the set of switches. All switches are available and support aggregation, and all links have a constant rate of 1. Servers are connected only to leaf switches. Such a topology can be viewed as if the leaf switches are effectively top-of-rack (ToR) switches in a small datacenter topology, where each rack accommodates a distinct number of servers (or VMs). Fig. 2 provides an illustration of the network and the load being handled. Each leaf switch is connected to a rack of several servers where the number of servers in the rack is marked in the gray square depicting the rack. In particular, the load handled by the four leaf switches is $(2, 6, 5, 4)$ (from left to right). In our example, the maximum number of blue switches allowed is set to $k = 2$. Each link e is marked with the utilization cost of this link, $\mathrm{msg}_e(T, L, U) \cdot 1$.

Figs. (2a), (2b), and (2c) show the results of applying strategies Top, Max, and Level, respectively, to such a network and load. The optimal approach, which is obtained by our proposed algorithm, SOAR (formally described and analyzed in Sec. 4), ends up picking a non-trivial set of blue nodes as can be seen in Fig. (2d). This allocation strictly outperforms all three contending strategies.

Fig. 3 provides examples of the optimal sets of blue nodes for increasing values of k. We note that in general, optimal solutions need not be unique, and for such cases ($k = 1, 4$, in Figs. (3a) and (3d)) we provide one of these solutions. However, for some cases ($k = 2, 3$, in Figs. (3b) and (3c)) the optimal solutions are unique. Considering the specific optimal solutions provided for these cases, we observe that the optimal sets of blue nodes, for increasing values of k, are not necessarily monotone. Namely, adding even one more blue node to the set can change the set of blue nodes completely.

4 SOAR: AN OPTIMAL ALGORITHM

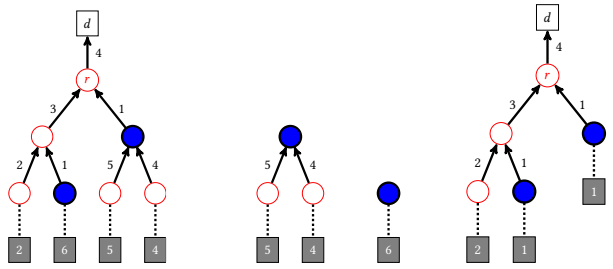
> "Those who sow in tears will reap with songs of joy."
>
> Psalm 126:5

In this section we describe our algorithm, SOAR, that produces an optimal solution to the ϕ-*BIC* problem.[2] The intuition underlying our algorithm is that if we are able to optimally sow blue nodes in the right locations, then it is possible to reap significantly improved performance in terms of the system's utilization complexity. The main technical contribution of the paper is the following theorem (we defer the formal analysis and proof to Sec. 6).

Theorem 4.1. *Given a weighted tree network T, rates ω, a load L, availability Λ, and a bound k on the number of allowed blue switches, algorithm SOAR solves the ϕ-BIC problem in time $O(n \cdot h(T) \cdot k^2)$.*

Before describing the algorithm we first provide some insight as to the structure induced by any solution, and the long-range effect of having a sequence of red nodes along a path. These serve to provide a better understanding of our objective function $\phi(T, L, U)$ capturing the utilization cost of the system.

[2] SOAR stands for SOw-And-Reap.



(a) Network T **(b) Tree decomposition of T**

Figure 4: Example of the barrier perspective using tree-decomposition. The settings are the same as in Fig. 2.

4.1 Re-formulating the Utilization Complexity: A Barrier Perspective

When considering the ϕ-*BIC* problem, one can view any solution U as inducing a *tree partitioning*, such that while scanning the nodes from the leaves towards the root, for every blue node $v \in U$ that has no blue nodes in its subtree, we can *detach* the subtree rooted at v from the tree. Such a blue node v then becomes a leaf in the remaining tree, where its load is set to 1 in that tree. The overall utilization cost in the original tree is simply the sum of utilization costs in all the subtrees thus produced. The reason for this equality is that every blue node in the tree effectively forms a *barrier* between the subtree rooted at the node, and the remaining tree above the node. Fig. 4 provides an illustration of such a decomposition, and the breakdown of the utilization complexity as the sum of the utilization complexity over the subtrees. Note that in each subtree, each node that was blue in the original network T is either a leaf (with load 1), or a destination.

An alternative way to view the system's utilization complexity, which serves as the fulcrum in our proposed algorithm, SOAR, is considering the distance of any node v from its *closest* blue ancestor (or the destination d, if no such blue ancestor exists). Formally, the next lemma, which follows directly from the definition of ϕ in Eq. 1, provides an alternative characterization of $\phi(T, L, U)$.

Lemma 4.2. *Consider a tree network T with load L, and consider any set $U \subseteq T$ of blue nodes. For each node $v \in T$ let $p_v^* = p_v^*(T, U)$ denote v's closest blue ancestor, if one exists, or d, otherwise. Then,*

$$\phi(T, L, U) = \sum_{v \in U} 1 \cdot \rho(v, p_v^*) + \sum_{v \notin U} L(v) \cdot \rho(v, p_v^*). \tag{3}$$

To illustrate Eq. (3), consider Fig. (3b). By Eq. (1), the sum over all edges (from left to right, bottom-up) is $(2+1+5+4)+(3+1)+4 = 20$. Alternatively, by Eq. (3) (considering the relevant nodes from left to right, bottom-up) is $(3+2)+(2\cdot 3+5\cdot 1+4\cdot 1) = 20$.

We note that the closest blue ancestor of a node is equivalent to the blue node serving as a barrier in our description of the tree-decomposition induced by any set of blue nodes U.

The alternative formulation of our objective in Eq. 3 lies at the core of our proposed algorithm, SOAR. In particular, as we show in the sequel, this formulation will serve to evaluate the potential effect, in terms of utilization, of having a node colored red or blue.

Algorithm 2 SOAR(T, L, Λ, k)

Require: A tree T, load L, availability Λ, k # of blue nodes
Ensure: ϕ-$BIC(T, L, \Lambda, k)$
1: run SOAR-Gather(T, L, Λ, k) at each node v
▷ gather X_v and Y_v in bottom-up order
2: wait until destination receives X_r
3: run SOAR-Color(k) at each node v
▷ determine node colors in top-down order

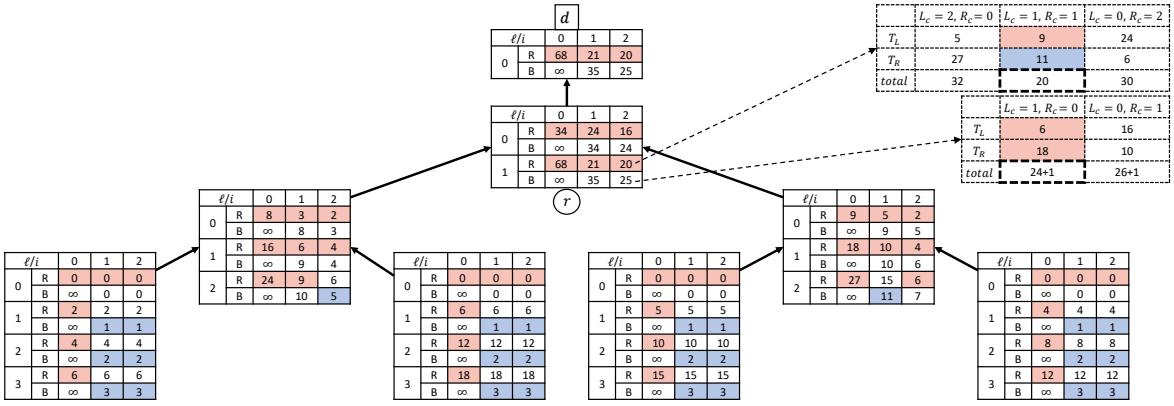
4.2 Overview of SOAR

In this section we provide a high-level overview of SOAR, formally defined in Algorithm 2, which solves the ϕ-*BIC* problem. Our solution is based on dynamic programming, and is split into two phases. In the first phase we apply algorithm SOAR-Gather, formally defined in Algorithm 3 (in Sec. 6), for gathering the information required for computing an optimal solution. This is followed by the second phase where we apply algorithm SOAR-Color, formally defined in Algorithm 4 (in Sec. 6), which traces back the actual allocation of blue nodes along the breadcrumbs produced in the first phase. We now provide further details as to each of the phases, and discuss their design criteria.

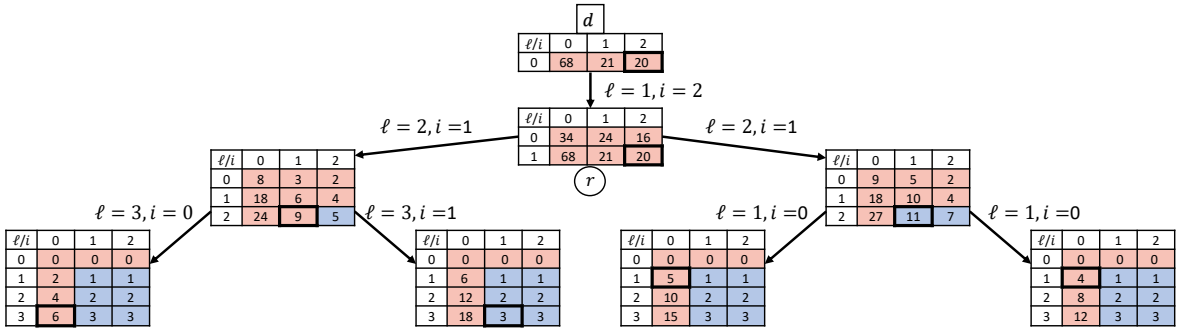
SOAR-Gather. Algorithm SOAR-Gather, which effectively builds the dynamic programming table, uses parameterized potential functions, which account for the long-range effect of having red nodes in the subtree rooted at some given node. In particular, a parameter ℓ used in these potential functions corresponds to the possible distance of a node in the network from the closest blue ancestor (or the destination d, if there is no such ancestor) in the network. Since we don't know the coloring at this phase, we compute the potential function for node v and each possible value ℓ between zero and $D(v)$. The key observation that enables us to do this efficiently is that conditioning on a parent v having some specific color (either red or blue) and the value of parameter ℓ, the subtree T_v can be independently optimized from the rest of the tree.

The information gathered during the first phase provides the breadcrumbs required for determining the allocation of blue nodes within the network in the second phase. In particular, each node gathers two sets of values: (i) X_v, which prescribes the utilization that would potentially be added in links further up in the tree (for any amount i of blue nodes being distributed in the subtree rooted at v) until the closest blue ancestor (at distance ℓ from v), and (ii) Y_v, which registers the distinct partitioning of blue nodes to children of v, for any combination of number of blue nodes i that should be distributed in the subtree rooted at v, any distance ℓ from v to its closest blue ancestor, and for both cases of whether v is blue or red. As we show in our proofs in Sec. 6, these partitions can be computed efficiently. In SOAR-Gather information is gathered while scanning the nodes of the network from the leaves upwards.

SOAR-Color. Algorithm SOAR-Color, which determines the color of each node, either blue or red, traces an optimal path in the dynamic programming table calculated by SOAR-Gather. Initially, a switch is set to be red, and this is altered only if setting the switch to being blue implies a smaller value of the potential function



(a) SOAR-Gather running example.



(b) SOAR-Color running example.

Figure 5: SOAR running example

computed during the first phase, for the specific parameter corresponding to the distance of that node from its nearest blue ancestor, or from the destination d (in case the node has no blue ancestor). The nodes are assigned colors in this manner while scanning them from the root downwards, where each node further alerts each of its children as to the number of blue nodes that should be distributed in the subtree rooted at that child. The number of blue nodes assigned to the subtree rooted at each child, i.e., the partitioning of blue nodes across the subtrees of the node, is also available as one of the outputs of SOAR-Gather.

We note that both SOAR-Gather and SOAR-Color are described as distributed, asynchronous, algorithms, where synchronization between nodes is maintained by waiting for specific messages to be received from either the children of a node, or the parent of a node. For SOAR-Gather, the leaves of the tree network T initiate the messages carrying information upwards in the network towards the destination server d. For SOAR-Color, the destination server d initiates the flow of information, by sending the bound on the number of allowed blue switches, k, to the root node r.

4.3 SOAR Running Example

In this section we demonstrate a running example of SOAR. We show how the optimal solution in Fig. (2d) was obtained. Figure (5a) presents the data structures that SOAR manages in each node during the SOAR-Gather phase. Every node v maintain a table with three dimensions, denoting possible parameters of the potential function: (i) The number i of possible blue nodes in its subtree, ranging from 0 to k (columns), (ii) the possible distance ℓ from its *closest blue ancestor*, if one exists, or d, otherwise (rows), and (iii) the node *color* (R or B). Fig. (5b) shows the way in which SOAR-Color tracks an optimal path along the tables computed during the SOAR-Gather phase.

The SOAR-Gather phase: For every node, and every combination of these parameters, the table maintains the minimal total utilization that would be incurred by the subtree T_v. This includes the effect of the color of v on the utilization of links above v (ℓ levels up). These tables are calculated during the SOAR-Gather phase of the algorithm which proceeds from the leaves towards the root, where each node calculates the values based on the tables available at its children. Note that a node reports only one value, for every combination of i and ℓ, taken as the *minimum* of being red or blue (the minimum appears with its appropriate color in Fig. (5a)).

For example, consider the root r and how r calculates the entry for $\ell = 1$ (i.e., it is at distance 1 from its nearest blue ancestor, or d – where in this case it is its distance from d) and it has $i = 2$ blue nodes to distribute within its subtree. Assume its children calculated their tables correctly.

First, r considers the case where it is colored red (R) (top dotted table in Fig. (5a)). In this case, it has 2 blue nodes to distribute in the subtrees rooted at its children. Therefore it takes the minimum of three possible cases, $(L_C = 2, R_C = 0), (L_C = 1, R_C = 1)$ or $(L_C = 0, R_C = 2)$, where L_C and R_C denote the number of blue nodes to be used in the left and right child of r, respectively. By checking the tables of its left and right children, r can find that the minimum is obtained in the case where $(L_C = 1, R_C = 1)$. where its left child will be colored red and will contribute 9 to the overall utilization, and the right child will be colored blue and will contribute 11 to the overall utilization. When checking its children tables r considers the entries corresponding to $\ell = 2$ since r is assumed to be red, and its own value of ℓ is 1. So overall T_r will contribute 20 to the utilization under these settings.

The other alternative is the case where r is colored blue (bottom dotted table in Fig. (5a)). This leads to only two possible partitions of remaining blue nodes across the subtrees rooted at its children, since r has already "used-up" one of the $i = 2$ blue nodes available in its subtree. The minimum configuration is when the left child of r gets to distribute the remaining blue node, $(L_C = 1, R_C = 0)$, in which case the subtree rooted at this child contributes 6 to the overall utilization (note that we consider the child's table for $\ell = 1$, since the child is at distance 1 from its closest blue ancestor, being the root in this case). The right child gets to distribute no blue nodes, and thus contributes 18 to the overall utilization. The utilization contributed by T_r thus totals 25, where $6 + 18 = 24$ are due to the subtrees rooted at the children of r, and 1 more contributed by r since its distance to d is one and r is assumed to be blue. Taking the minimum of 20 and 25, r will report to its parent that for this setting, $(\ell = 1, k = 2)$, T_r will contribute 20 to the utilization, and its color will be red.

The SOAR-Color phase: In the next phase, the coloring is done using SOAR-Color by tracing an optimal path over the tables generated during the SOAR-Gather phase, from the root to leaves. For our example, the destination, d, needs to place $k = 2$ blue nodes in the *network*. The utilization in this case is 20. d passes the values $\ell = 1$ and $i = 2$ (from which the minimal utilization was derived) to its child, r. At this point r looks up the color corresponding to these values in its table, and determines its own color, red in this case. Furthermore, r knows the number of blue nodes available for distribution in its subtree, and its distance from its closest blue ancestor (or d, in this case). r can then determine the amount of blue nodes it needs to pass on to each of its children, for distribution in their subtrees. With this information it can recursively determine the color of its children. Fig. (5b) shows the color and configuration that was selected at each node (bold square) in the optimal solution provided by SOAR.

4.4 SOAR: Practical Aspects and Limitations

While SOAR minimizes the network utilization, namely the overall transmission time over all links, using bounded in-network computing, there are various *practical* aspects that are related to the implementation of our approach, and the benefits it provides. For example, how and where should switches store the aggregated values? What are the effects of packet-loss and latency (affecting the delivery of messages)? For line-rate aggregation, what synchronization mechanisms are required? How can one use our solution in a system handling multiple tenants and workloads, and what would be the overhead of using our approach?

For the most part, these questions are applicable to most in-network computing environments performing in-network aggregation. However, some aspects are specifically more pronounced in our model, namely, the distinction between aggregating nodes (which wait for all incoming information before forwarding a message), and non-aggregating nodes, which simply follow a store-and-forward regime. We plan to study these specific aspects in future work, as we note that these may well affect the performance whenever significant variable delay is manifested (e.g., by inducing an overall lower rate of information flow), and for some aggregating functions the memory tolls may be non-negligible.

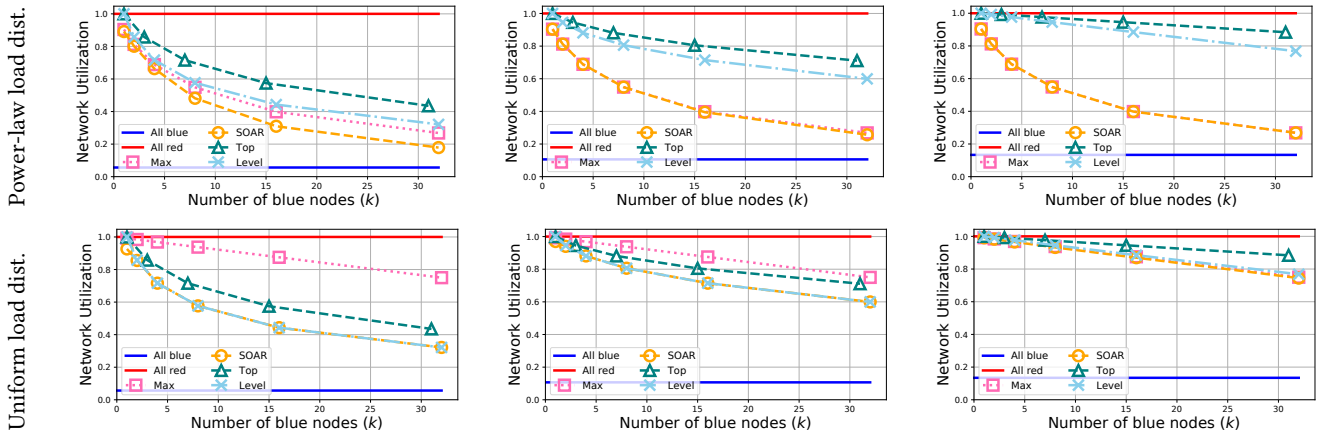
Another significant aspect related to our approach is the fact that our model assumes that any message being transmitted throughout the system is of size at most M. For some aggregation functions (e.g., bitwise-functions, or max/min), assuming such a bound is quite reasonable as it can be determined by the maximum size of a message generated by the servers. The optimality of SOAR relies on this assumption. However, for other functions, performing aggregation, and furthermore doing so repeatedly, might result in a message size increase that may be proportional to, or at least monotone with, the size of the workload (e.g., sum or product functions). In such cases, *SOAR is not guaranteed to ensure optimal performance*. However, we do evaluate such effects in Sec. A, where we study the performance of SOAR in terms of the overall number of *bytes* being transmitted (which essentially take into account the effect of increasing message size while doing in-network aggregation). Our results show that in some cases, the decisions of SOAR allow it to come close to a *lower bound* on the optimal performance possible.

Bearing the above limitations in mind, one should note that practical solutions that address various of the above issues, are already being deployed in real systems and datacenters (e.g., Nvidia's SHARP [21] protocol). However, we are not aware of any such solutions which handle bounded in-network computing capabilities as in our model, nor of any solutions that are optimized for multiple workload. Bridging the gaps between our model and solutions, and real-life deployments, remains a significant challenge.

5 EVALUATION

In this section we describe the results of our evaluation of SOAR, where we performed extensive simulations which provide further insight as to its performance. In our evaluation, we examine both the utilization complexity induced by SOAR (and at times additional contending strategies). We also consider the *byte complexity* which, is the actual network load, in bytes, imposed by performing the Reduce operation, for real-life applications (due to space constraints, these results appear in the appendix).

Most of our evaluation makes use of the following system characteristics (unless explicitly stated otherwise). We consider complete binary weighted trees as the underlying network, denoted by $\mathrm{BT}(n)$, where n is the number of nodes in the network, including the destination server. We allow non-zero load to be placed only in the leaves of the tree. These leaves serve as top-of-rack switches connected to



(a) constant ($\omega = 1$) **(b) linear increasing ($\omega = i$)** **(c) exponentially increasing ($\omega = 2^i$)**

Figure 6: SOAR vs. other strategies for distinct schemes of rates (Fig. 6a-6c), and distinct load distributions (power-law in the top plot, uniform in the bottom plot).

servers that generate load, whereas the remaining network serves to model the higher levels of a datacenter network which facilitates the flow of information from the various worker servers, to the destination server which acts as the aggregator. We consider two distributions for the load at the leaves of the network: (i) uniform, where the integer load of each node is picked u.a.r. in some range $[x, y]$, and (ii) power-law, where the integer load of each node is picked from a power-law distribution. The distributions characteristics are as follows; The mean of both distributions is 5, the variance is 0.65625 and 97.1 for the uniform and the power-law, respectively. The (min, max) values are $(4, 6)$ and $(1, 63)$ for the uniform and the power-law distributions, respectively. We consider three different rate schemes: (i) constant rates, where all link rates are equal to 1, (ii) linear rates, were $\omega(e)$ increases linearly, by adding 1, from leaf edges (rate 1) towards the root, and (iii) exponential rates, were $\omega(e)$ increases by doubling (i.e., a power of 2), from leaf edges (rate 1) towards the root. Each experiment was repeated ten times, and we present the average performance for each such set of experiments. For clarity, we present error bars only where we encountered significant variance in the results.

In most of our results, we present the *normalized* performance of an algorithm, where normalization is usually with respect to the *all-red* scenario. This essentially shows the cost reduction of the specific scenario, compared to the *all-red* solution. I.e., if the performance of an algorithm is $\alpha \in [0, 1]$ in some scenario, this means that the algorithm incurs an α fraction of the cost of the all-red solution when preforming Reduce in that scenario.

5.1 Comparing SOAR with Other Strategies

In this section we consider the performance of SOAR compared to the performance of several contending strategies for solving the ϕ-*BIC* problem. Specifically, we focus our attention on the simple strategies described in our motivating example in Sec. 3, namely, (i) *Top*, (ii) *Max*, and (iii) *Level*.

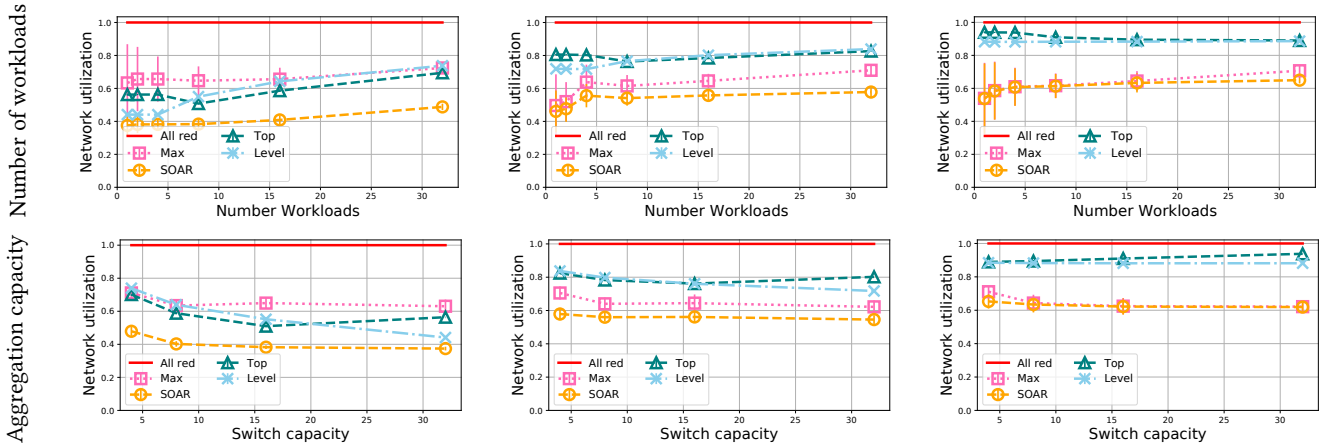
Fig. 6 presents the performance of SOAR alongside the performance of the contending strategies in distinct rate regimes (subfigures 6a-6c), for different workload distribution (top and bottom), using BT(256). We consider distinct values of $k = 1, 2, 4, 8, 16, 32$, and performance is normalized to the all-red strategy. We further plot the performance of the *all-blue* solution for reference. As would be expected, all strategies exhibit improved performance for increasing values of k, which allows for more in-network aggregation, translating to reduced utilization complexity.

Since SOAR is optimal, it exhibits the best performance in all scenarios. This serves to show that using SOAR ensures robustness regardless of load distribution or link rates. However, the second-best strategy strongly depends on the load distribution, and the link rates. The power-law load distribution favors the Max strategy, since high-load leaf-switches that perform aggregation induce a significant reduction in overall utilization complexity. For the uniform distribution, however, the Level strategy fares best, since it implies load balancing the uniform loads at the leaf-switches throughout the network. For such scenarios, the Level strategy essentially mimics the "barrier" approach underlying the design of SOAR, as described in Sec. 4.1. The Top strategy is the most sensitive to the link rates, where having higher rates towards the root of the network implies that performing in-network aggregation further up provides far lesser benefits than doing so closer to the leaves (or closer to the middle of the network).

Takeaways: SOAR can significantly outperform other strategies across different workloads and link rates functions. A small fraction of nodes with in-network processing capabilities is enough to reduce network utilization substantially.

5.2 Multiple Workloads

In this subsection we consider the problem of handling multiple workloads, and determining where aggregation should take place for each such workload. Each workload is determined by its L_t, $t = 0, 1, 2, \ldots$. We consider the workloads as arriving in an *online*



(a) constant ($\omega = 1$) **(b) linearly increasing ($\omega = i$)** **(c) exponentially increasing ($\omega = 2^i$)**

Figure 7: SOAR vs. other strategies when aggregating multiple workloads online. Subfigures represent distinct schemes of rates (Fig. 7a-7c), and the effect of increasing the number of workloads (for switch capacity 4, top plots), and increasing the aggregation capacity (for 32 workloads, bottom plots).

fashion, such that determining the aggregating switches for workload L_t should be settled before handling workload L_{t+1}. We assume each switch s has a predetermined *aggregation capacity* $a(s)$ which bounds the number of workloads for which s can be assigned as an aggregating switch. We let $a_t(s)$ denote the *residual aggregation capacity* available at s before handling workload L_t. If switch s is designated as an aggregation switch when handling workload L_t, then $a_{t+1}(s) = a_t(s) - 1$, and $a_{t+1}(s) = a_t(s)$ otherwise.

We consider the performance of the various strategies used in Sec. 5.1, when applied repeatedly to the sequence of workloads $L_0, L_1, \ldots$, given as input. The set of switches available for aggregation when handling workload L_t is defined by $\Lambda_t = \{s \mid a_t(s) > 0\}$.

We generate our sequence of workloads in an online fashion, by drawing each workload from either the uniform load distribution, or the power-law load distribution, each with probability 1/2.

In our evaluation we consider the effect of varying the aggregation capacity, and the number of workloads. As a baseline we consider the topology BT(256), with $k = 16$, $a(s) = 4$ for every switch s, and 32 workloads. Fig. 7 shows the performance of SOAR compared to the performance of the various strategies described in Sec. 3. Similarly to our results presented in Sec. 5.1, our evaluation considers 3 scaling laws for link rates: constant (in Fig. 7a), linearly increasing (in Fig. 7b), and exponentially increasing (in Fig. 7c).

When considering the effect of handling more workloads (Top plot in each column), the normalized utilization ratio (compared to that of the all-red solution) tends to increase as we handle more workloads, and the improved performance demonstrated by SOAR compared to the best contending strategy is more pronounced as the weight differences across layers are smaller. We note that as the number of workloads increases, the performance would converge to that of the all-red configuration, regardless of the strategy being used. This follows from the fact that the aggregation capacity is bounded, implying that once the number of workloads is large enough, further workloads cannot benefit from any aggregation, and the initial benefits of aggregating the prefix of the workload arrival sequence become marginal compared to the toll imposed by the entire sequence.

It is instructive to note that the second-best strategy varies significantly, where for exponentially increasing rates the performance of the Max strategy is closest to that of SOAR while for constant rates either the performance of the Level strategy or the Top strategy comes closest to that of SOAR.

When considering the effect of increasing the aggregation capacity at each switch, one can see that most strategies exhibit improved performance as the aggregation capacity increases, and SOAR exhibits the best performance across all scenarios, where the differences are again more pronounced as the differences in rates across levels is smaller. An exception to this performance is exhibited by the Top strategy, which actually fares worse as aggregation capacity increases. This is due to the fact that the larger capacity enables the strategy to handle more workloads closer to the root, which accentuates its sub-optimality. Finally, we note that when aggregation capacity is unbounded, SOAR will produce the optimal solution possible (for any given k) *even in the online setting*, since it is optimal for every workload, and workloads are handled separately and independently.

Takeaways, SOAR exhibits the best performance compared to other strategies in the online settings (although it is not proven to be optimal). Furthermore, for small switch capacity and many workloads, SOAR obtains more considerable gains.

5.3 SOAR Run-Time Evaluation

In this section we evaluate the running-times of SOAR, SOAR-Gather and SOAR-Color. We implemented our simulation in python 3.8 and the evaluation was done on a laptop equipped with an Intel core i7(10875H) CPU and 32GB of RAM.

When measuring the running-time of SOAR-Gather and SOAR-Color we conclude that the running time of SOAR-Color is faster

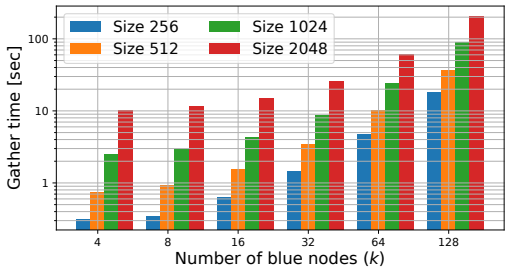


Figure 8: SOAR running time

by three orders of magnitude compared to SOAR-Gather; While SOAR-Gather runs in seconds, SOAR-Color runs in milliseconds.

In Fig 8 we present the average running time of SOAR-Gather over ten experiments for different network sizes and values k. The figure shows the running time in seconds, for $k = 4, 8, 16, 32, 64$, and 128, and for network sizes 256, 512, 1024, and 2048. Following Theorem 4.1, we can observe that the running time is indeed quadratic in k and close to linear in n (where our results prove an upper bound of $n \log n$). We note Fig. 8 is in log-log scale.

We highlight that the above results apply to a *serial and centralized* implementation of SOAR-Gather on a single host, which may require up to a few minutes for a relatively large network (2048 switches) and many aggregating nodes (128). For moderately sized networks, or when deploying far fewer switches, the running time of SOAR-Gather is on the order of tens of seconds, or less. We further note that SOAR-Gather can also be implemented in a *parallel or distributed* manner (along a parallel DFS-scan from leaves to the root), which would result in a significant speedup, while requiring more computing power to be used in parallel. We leave this topic for future work.

6 ANALYSIS OF SOAR

6.1 Notation and Definitions

We first introduce some notation that would be used throughout our proofs. For every node v, we let $c_1, \ldots, c_{C(v)}$ denote the children of v (in some arbitrary fixed order). For every $m = 1, \ldots, C(v)$ we let T_v^m denote the subtree rooted at v containing only the subtrees rooted at children $c_1, \ldots, c_m$, and let $\tilde{T}_v^m$ denote the *extended subtree* of T_v^m, which is extended by adding the link $(v, p(v))$. We further let $T_v = T_v^{C(v)}$ denote the subtree rooted at v (containing all subtrees of all children of v), and let $\tilde{T}_v$ be the extended subtree of T_v. For a node v and $\ell \le D(v)$, let A_v^ℓ be the ancestor at distance ℓ from v.

For every node v and every $m = 1, \ldots, C(v)$, given any $\ell = 0, \ldots, D(v)$ and any set of blue nodes $U \subseteq T_v^m$, we consider the (v, m)-*potential* of ℓ and U, $\pi_v^m(\ell, U)$, defined by

$$\pi_v^m(\ell, U) = \left(\sum_{e \in T_v^m} \mathrm{msg}_e(T_v^m, L, U) \cdot \rho(e) \right) + \mathrm{msg}_{(v,p(v))}(\tilde{T}_v^m, L, U) \cdot \rho(v, A_v^\ell). \quad (4)$$

and we further use $\pi_v(\ell, U)$ to denote $\pi_v^{C(v)}(\ell, U)$. We note that the (v, m)-potential is only defined for $U \subseteq T_v^m$. For ease of notation, we will omit this explicit requirement in the remainder of this section. Furthermore, if v is a leaf we take $C(v) = 0$ which results in having

$$\pi_v^0(\ell, U) = \mathrm{msg}_{(v,p(v))}(\tilde{T}_v^0, L, U) \cdot \rho(v, A_v^\ell), \quad (5)$$

Lastly, we note that by the definition of π_v^m in Eq. (4), and the definition of ϕ in Eq. (1) we have

$$\phi(T, L, U) = \pi_d(0, U) = \pi_r(1, U). \quad (6)$$

A key property of our potential function, that follows from the above definitions, is that, *conditioning* on the color of v, we can decompose the calculation of $\pi_v^m(\ell, U)$ into calculating π for smaller problems, as depicted by the following lemma, whose proof is omitted due to space constraints, and can be found in [45].

Lemma 6.1. *If $v \in T$ is a non leaf node and U is the set of blue nodes, then*

$$\pi_v(\ell, U) = \sum_{m=1}^{C(v)} \pi_{c_m}(1, U) + 1 \cdot \rho(v, A_v^\ell) \quad \text{if } v \in U \quad (7)$$

$$\pi_v(\ell, U) = \sum_{m=1}^{C(v)} \pi_{c_m}(\ell + 1, U) + \rho(v, A_v^\ell) \cdot L(v) \quad \text{if } v \notin U. \quad (8)$$

6.2 Optimality of SOAR

We first consider algorithm SOAR-Gather, which gathers the required information about *potentially optimal* configurations. The algorithm essentially scans the tree from the leaves towards the root, varifying in every node along the scan the overall message cost of the subtree rooted at that node, assuming it would end up allocating i blue nodes within the subtree, for all possible values of i. In doing so, the algorithm evaluates these costs while taking into account all the possible distances of the node being considered (captured by parameter ℓ) from the nearest blue ancestor or d.

The intuition behind this approach follows from Lemma 4.2 and the partitioning-view of the solution that would eventually be produced; The following lemma serves as the main technical tool for proving the correctness of SOAR.

Lemma 6.2. *For every node v, every $m = 1, \ldots, C(v)$, every $\ell = 0, \ldots, D(v)$, and every $i = 0, \ldots, k$, if v is not a leaf then Y_v^m as computed by SOAR-Gather satisfies*

$$Y_v^m(\ell, i, R) = \min_{|U|=i, v \notin U} \pi_v^m(\ell, U) \quad (9)$$

$$Y_v^m(\ell, i, B) = \min_{|U|=i, v \in U} \pi_v^m(\ell, U). \quad (10)$$

Furthermore, for every node v, every $\ell = 0, \ldots, D(v)$, and every $i = 0, \ldots, k$, $X_v(\ell, i)$ as computed by SOAR-Gather satisfies

$$X_v(\ell, i) = \min_{|U|=i} \pi_v(\ell, U). \quad (11)$$

The lemma follows from a double induction argument on the height of the subtree T_v rooted at any node v, and indices $m = 1, \ldots, C(v)$ of the children of a node v. The proof is omitted due to space constraints, and can be found in [45].

Lemma 6.2 ensures that the values gathered and computed by the nodes while running SOAR-Gather indeed correspond to the configurations minimizing $\pi_v(\ell, U)$. In particular, by Eq. (6), the

[3] When $i = 0$ then $Y_v^m(\ell, i, B) = \infty$.

Algorithm 3 SOAR-Gather(T, L, Λ, k) at node v

Require: A tree T,load L, availability Λ, k # of blue nodes
Ensure: Correct potential functions, X_v, Y_v, at each node v

1: **if** v is a leaf node **then**
2: **for** $\ell = 0, \ldots, D(v)$ **do** ▷ $D(v)$ distance of v from root
3: $X_v(\ell, 0) = \rho(v, A_v^\ell) \cdot L(v)$
4: **for** $i = 1, \ldots, k$ **do**
5: **if** $v \in \Lambda$ **then** ▷ v has available capacity
6: $X_v(\ell, i) = \rho(v, A_v^\ell)$ ▷ v is blue
7: **else**
8: $X_v(\ell, i) = \rho(v, A_v^\ell) \cdot L(v)$ ▷ v is red
9: send X_v to $p(v)$ and **return** ▷ inform parent
10: wait to receive X_c from each child c of v
11: **for** $m = 1, \ldots, C(v)$ **do**
12: **for** $\ell = 0, \ldots, D(v)$ **do**
13: **for** $i = 0, \ldots, k$ **do**
14: **if** $m = 1$ **then**
15: **if** $v \in \Lambda$ **then**
16: $Y_v^m(\ell, i, B) = X_{c_m}(1, i-1) + \rho(v, A_v^\ell)$ [3]
17: **else**
18: $Y_v^m(\ell, i, B) = \infty$
19: $Y_v^m(\ell, i, R) = X_{c_m}(\ell+1, i) + \rho(v, A_v^\ell) \cdot L(v)$
20: **else**
21: **if** $v \in \Lambda$ **then**
22: $Y_v^m(\ell, i, B) = \text{mCost}(\ell, i, Y_v^{m-1}, X_{c_m}, B)$ [3]
23: **else**
24: $Y_v^m(\ell, i, B) = \infty$
25: $Y_v^m(\ell, i, R) = \text{mCost}(\ell, i, Y_v^{m-1}, X_{c_m}, R)$
26: **for** $\ell = 0, \ldots, D(v)$ **do**
27: **for** $i = 0, \ldots, k$ **do**
28: $X_v(\ell, i) = \min\left\{Y_v^{C(v)}(\ell, i, B), Y_v^{C(v)}(\ell, i, R)\right\}$
29: send X_v to $p(v)$ and **return**

30: **procedure** mCost$(\ell, i, Y_v^{m-1}, X_{c_m}, \text{color})$
31: **if** color == B **then**
32: **return** $\min_{0 \le j < i} \left[Y_v^{m-1}(\ell, i-j, B) + X_{c_m}(1, j)\right]$
33: **else** ▷ color == R
34: **return** $\min_{0 \le j \le i} \left[Y_v^{m-1}(\ell, i-j, R) + X_{c_m}(\ell+1, j)\right]$

lemma guarantees that the value computed for $X_d(0, k)$, where d is the destination server, is indeed the minimal utilization cost possible using k blue nodes.

In the second phase of SOAR, SOAR-Color essentially traces back the allocation of blue nodes along the optimal path in the dynamic programming performed by SOAR-Gather. The following lemma shows that SOAR-Color indeed produces an optimal solution to the ϕ-*BIC* problem (proof is omitted and appears in [45]).

Lemma 6.3. *The set of blue nodes U determined by SOAR-Color minimizes the utilization complexity, and $|U| \le k$.*

The proof of Theorem 4.1 now follows immediately from combining Lemma 6.2 and Lemma 6.3.

Algorithm 4 SOAR-Color(k) at node v

Require: X_v, Y_v
Ensure: Optimal coloring

1: **if** v is destination **then**
2: send $(k, 1)$ to r
3: color v red and wait for (i, ℓ^*) from $p(v)$
▷ ℓ^*: distance from root or closest blue ancestor
▷ i: number of blue noes in T_v
4: **if** v is a leaf node and $i > 0$ **then**
5: color v blue and **return**
6: **if** $Y_v^{C(v)}(\ell^*, i, B) < Y_v^{C(v)}(\ell^*, i, R)$ **then**
7: color v blue
8: $\ell^* = 0$ ▷ reset distance to closest blue ancestor
9: **for** $m = C(v), \ldots, 2$ **do** ▷ children in reverse order
10: $j = \text{mSplit}(\ell^*+1, i, Y_v^{m-1}, X_{c_m}, \text{color of } v)$
11: send (j, ℓ^*+1) to c_m
12: $i = i - j$
13: **if** v is blue **then** ▷ handle c_1 last
14: send $(i-1, \ell^*+1)$ to c_1
15: **else**
16: send (i, ℓ^*+1) to c_1
17: **return**

18: **procedure** mSplit$(\ell, i, Y_v^{m-1}, X_{c_m}, \text{color})$
19: **if** color == B **then**
20: **return** $\arg\min_{0 \le j < i} \left[Y_v^{m-1}(\ell, i-j, B) + X_{c_m}(1, j)\right]$
21: **else** ▷ color == R
22: **return** $\arg\min_{0 \le j \le i} \left[Y_v^{m-1}(\ell, i-j, R) + X_{c_m}(\ell+1, j)\right]$

Proof of Theorem 4.1. The correctness of the algorithm follows from Lemma 6.2 and Lemma 6.3. For the running time of SOAR, we note that it is dominated by the running time of SOAR-Gather, which, in turn, is dominated by the for-loop in lines 12-25. This loop is performed once for every edge $(v, p(v))$. This gives an overall running time of $O(n \cdot h(T) \cdot k)$ for this loop over all edges, where in each iteration the mCost procedure is performed at most once, implying an overall running time of $O(n \cdot h(T) \cdot k^2)$. The result follows. □

7 RELATED WORK

Data aggregation has been studied extensively in various contexts [23], where significant focus was given to wireless sensor networks [38], alongside scheduling algorithms for optimizing the induced convergecast tree [35], and characterizing the type of functions that can be efficiently aggregated [23, 58].

MapReduce [15] has proven to be a fundamental paradigm for various applications in distributed environments. Aside from being a cornerstone of big data analytics, it is also being adopted and incorporated into additional applications and systems, such as large distributed databases, although at the expense of sometimes non-negligible complexity [58]. Significant efforts were made to improve the performance of MapReduce, including aspects related to scheduling [60], data placement [11], and data coding [29].

Efficiently performing distributed machine learning, and specifically the task of training deep neural networks, has been a fundamental concern in the past decade. In particular, network bottlenecks are arguably one of the major concerns when executing such tasks [28, 52]. Various methods for improving network performance and footprint in such systems have been proposed and implemented, including sparsification, quantization, and scheduling [16, 53, 57]. Furthermore, aspects pertaining to system (and network) heterogeneity and varying network topologies have also been shown to affect the performance of such systems [3, 54]. We refer to a recent survey of methods and strategies for optimizing networks for ML [39]. Of particular relevance to our work is the efficient scaling of distributed ML using a parameter server, which aggregates local computations, and distributes updated models during training [27]. Using this approach has been shown to provide significant improvements of ML training tasks, with an emphasis on reducing the network footprint of these tasks [28, 31, 33]. Furthermore, the advent of federated ML [8] has further increased the efforts of optimizing network performance for ML tasks.

In-network Computing (INC) has recently gained a lot of attention from researchers and industry alike [41, 43]. This paradigm is fueled by the ability to *program* the data plane, using, e.g., the P4 programming language [9], alongside advances in FPGA design and performance (including SmartNICs). Such devices, which enable performing non-trivial computation within the network elements themselves, with minimal effects on performance (e.g., throughput and latency) [17], are effectively deployed by large-scale providers [18]. Examples of such application logic implementations include MapReduce [10, 13, 34, 43], Paxos [6, 14, 24], ML [19, 44, 56], caching [25, 30], key-value stores [50], storage replication [26, 63], IoT data aggregation [32], compression [51], lock management [59], and packet-level ML [48]. Some more recent efforts target generalizing INC to arbitrary functionalities [61], most predominantly those related to network functions [46], and also studying aspects of energy efficiency of such solutions [49]. Similar efforts are being performed in HPC environments, with special emphasis on support for large-scale ML tasks (e.g., Nvidia's SHARP [21]). A recent work [7] also studied the problem of bounded resources in in-network computing, but the focus of the work was resource scheduling and not optimal placement as in our work. Whereas most of these works focus on the implementation of concrete functionalities within the network, we consider the orthogonal network-level problem of where should such capabilities be deployed, in order to optimize the cumulative system performance, regardless of the specific implementation and/or task to be performed.

8 DISCUSSION AND FUTURE WORK

This work considers the ϕ-*BIC* problem, where we need to determine the location of a limited number of aggregation switches within a tree network, so as to minimize the overall utilization complexity of reduce operations. This problem lays at the heart of many distributed computing use cases, most notably big data tasks using the MapReduce paradigm, and distributed and federated machine learning. Our work describes an optimal algorithm, SOAR, for solving the ϕ-*BIC* problem, and provides further insights as to the performance of SOAR via an extensive simulation study.

A future challenging task is to develop solutions that are applicable to *general* networks (i.e., not necessarily tree networks), thus supporting multi-path routing. Another interesting open problem is related to the multiple workloads scenario. The main question there is how to distribute the aggregation capacity available throughout the network to the various workloads being served. Specifically, every workload might be serviced by a *distinct* number of aggregation switches (i.e., there need not be a uniform k for all workloads).

Last but not least, we expect our approach to also be effective in designing algorithms that target minimizing the *delay* incurred by the system, or minimizing the load on bottleneck links, while using a bounded number of aggregation switches. However, our methodology may need to be modified significantly for such objectives, and may require new tools and insights. We conjecture, however, that these objectives – that of minimizing the overall utilization complexity, and that of minimizing the overall system delay or bottlenecks, are closely related, and a solution minimizing one of these objectives is expected to perform well also for the other objectives.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and our shepherd, Shay Vargaftik, for their valuable feedback which helped improve the paper. This project was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 864228 - AdjustNet).

REFERENCES

[1] Apache hadoop - mapreduce tutorial. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2021.
[2] Wikimedia downloads. https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles1.xml-p1p41242.bz2, 2021.
[3] Ahmed M. Abdelmoniem, Chen-Yu Ho, Pantelis Papageorgiou, Muhammad Bilal, and Marco Canini. On the impact of device and behavioral heterogeneity in federated learning, 2021. arXiv, https://arxiv.org/abs/2102.07500.
[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.
[5] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, pages 63–74, 2010.
[6] Giacomo Belocchi, Valeria Cardellini, Aniello Cammarano, and Giuseppe Bianchi. Paxos in the NIC: hardware acceleration of distributed consensus protocols. In *DRCN*, pages 1–6, 2020.
[7] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Switches for hire: resource scheduling for data center in-network computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 268–285, 2021.
[8] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konecný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *MLSys*, 2019.
[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.
[10] Valerio Bruschi, Marco Faltelli, Angelo Tulumello, Salvatore Pontarelli, Francesco Quaglia, and Giuseppe Bianchi. Offloading online MapReduce tasks with stateful programmable data planes. In *ICIN*, pages 17–22, 2020.
[11] Dazhao Cheng, Jia Rao, Yanfei Guo, Changjun Jiang, and Xiaobo Zhou. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. *IEEE Trans. Parallel Distributed Syst.*, 28(3):774–786, 2017.
[12] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, pages 98–109, 2011.

[13] Paolo Costa, Austin Donnelly, Antony I. T. Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *USENIX NSDI*, pages 29–42, 2012.

[14] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki-Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, 2020.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI*, pages 137–150, 2004.

[16] Aritra Dutta, El Houcine Bergou, Ahmed M. Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *AAAI*, pages 3817–3824, 2020.

[17] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, pages 345–362, 2019.

[18] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*, pages 51–66, 2018.

[19] Nadeen Gebara, Manya Ghobadi, and Costa Paolo. In-network aggregation for shared machine learning clusters. *Proceedings of Machine Learning and Systems (MLSys)*, 3, 2021.

[20] Takashi Goda. On the separability of multivariate functions. *Mathematics and Computers in Simulation*, 159:210–219, 2019.

[21] Richard L. Graham, Lion Levi, Devendar Bureddy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. Scalable hierarchical aggregation and reduction protocol (SHARP)^{TM} streaming-aggregation hardware design and evaluation. In *ISC*, pages 41–59, 2020.

[22] Sylvain Jeaugey. Massively scale your deep learning training with nccl 2.4, 2019. NVIDIA Developer Blog, https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[23] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *IEEE Commun. Surv. Tutorials*, 17(1):381–404, 2015.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. In *USENIX NSDI*, pages 35–49, 2018.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *SOSP*, pages 121–136, 2017.

[26] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX OSDI*, 2020.

[27] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, pages 583–598, 2014.

[28] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[29] Songze Li, Mohammad Ali Maddah-Ali, Qian Yu, and Amir Salman Avestimehr. A fundamental tradeoff between computation and communication in distributed computing. *IEEE Trans. Inf. Theory*, 64(1):109–128, 2018.

[30] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ASPLOS*, pages 795–809, 2017.

[31] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *SoCC*, pages 41–54, 2018.

[32] André Luiz R. Madureira, Francisco Renato Cavalcante Araújo, and Leobino N. Sampaio. On supporting iot data aggregation through programmable data planes. *Comput. Networks*, 177:107330, 2020.

[33] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In *USENIX HotCloud*, 2015.

[34] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter R. Pietzuch, and Alexander L. Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *CoNEXT*, pages 249–262, 2014.

[35] Baljeet Malhotra, Ioanis Nikolaidis, and Mario A. Nascimento. Aggregation convergecast scheduling in wireless sensor networks. *Wirel. Networks*, 17(2):319–335, 2011.

[36] James Murphy McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *Comput. Commun. Rev.*, 49(1):18–23, 2019.

[37] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *PODC*, pages 113–122, 2006.

[38] Eduardo Freire Nakamura, Antonio Alfredo Ferreira Loureiro, and Alejandro César Frery. Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Comput. Surv.*, 39(3):9, 2007.

[39] Shuo Ouyang, Dezun Dong, Yemao Xu, and Liquan Xiao. Communication optimization strategies for distributed deep neural network training: A survey. *J. Parallel Distributed Comput.*, 149:52–65, 2021.

[40] David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

[41] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *HotOS*, pages 209–215, 2019.

[42] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[43] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *HotNets*, pages 150–156, 2017.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation, 2019. arXiv, https://arxiv.org/abs/1903.06701.

[45] Raz Segal, Chen Avin, and Gabriel Scalosub. SOAR: Minimizing network utilization with bounded in-network computing, 2021. arXiv, https://arxiv.org/abs/2110.14224.

[46] Prateek Shantharama, Akhilesh S. Thyagaturu, and Martin Reisslein. Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies. *IEEE Access*, 8:132021–132085, 2020.

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

[48] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. Taurus: An intelligent data plane, 2020. arXiv, https://arxiv.org/abs/2002.08987.

[49] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *EuroSys*, pages 21:1–21:16, 2019.

[50] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. LaKe: The power of in-network computing. In *ReConFig*, 2018.

[51] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. ZipLine: in-network compression at line speed. In *CoNEXT*, pages 399–405, 2020.

[52] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. Network-accelerated distributed machine learning for multi-tenant settings. In *SoCC*, pages 447–461, 2020.

[53] Shuai Wang, Dan Li, and Jinkun Geng. Geryon: Accelerating distributed CNN training by network-level flow scheduling. In *INFOCOM*, pages 1678–1687, 2020.

[54] Shuai Wang, Dan Li, Jinkun Geng, Yue Gu, and Yang Cheng. Impact of network topology on the performance of dml: Theoretical analysis and practical factors. In *INFOCOM*, pages 1729–1737, 2019.

[55] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: incast congestion control for TCP in data-center networks. *IEEE/ACM Trans. Netw.*, 21(2):345–358, 2013.

[56] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *HotNets*, pages 25–33, 2019.

[57] Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, KAUST, 2020. http://hdl.handle.net/10754/662495.

[58] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

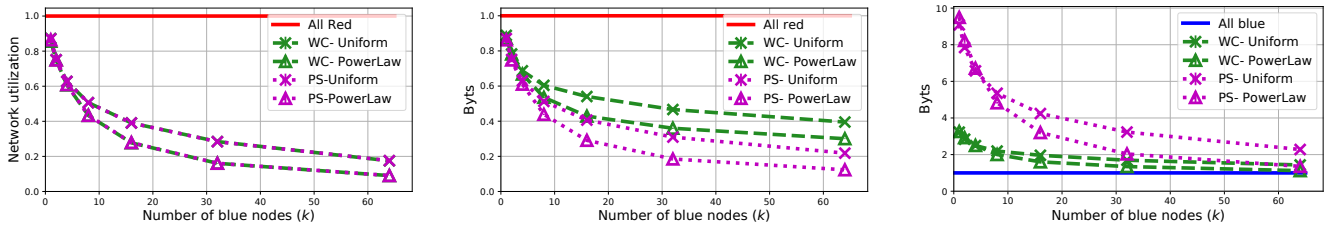
[59] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *SIGCOMM*, pages 126–138, 2020.

[60] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, pages 29–42, 2008.

[61] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *SIGCOMM*, pages 283–295, 2020.

[62] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *NetAI@SIGCOMM*, pages 8–13, 2020.

[63] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, 2019.



(a) Utilization **(b) Bytes (normalized to all-red)** **(c) Bytes (normalized to all-blue)**

Figure 9: SOAR performance for the WC and PS use cases.

A SOAR FOR DIFFERENT APPLICATIONS

In this section we present the results of our evaluation of SOAR for real-life applications, considering both the utilization complexity, as well as the *byte complexity*.

We consider two use cases for evaluating the system: (i) *big-data*, using a word-count task [1], where we make use of a Wikipedia dump [2], with an overall of 54M words, out of which 800K are unique. We refer to this use case as the *word count (WC)* use case. (ii) *distributed ML*, using distributed gradient aggregation with a parameter server [27], where worker servers independently perform neural-network training, over a 10K feature space, using 0.5 dropout rate [47][4], and send their updated gradients to a parameter server aggregating the information.[5] We refer to this use case as the *parameter server (PS)* use case.

We now turn to consider the performance of SOAR for distinct use cases, namely, WC, and PS. We focus our attention on the case of constant rates, which better emphasizes the differences in the performance. We distinguish between our utilization metric (which in the constant rate case is equivalent to the number of messages traversing the network), and the *byte complexity*, where we take into account the actual message size, and evaluate the overall number of bytes being transmitted throughout the network over all links. We note that our problem formulation, and our algorithm, do not target minimizing the byte complexity.

Fig. 9 shows the results of our evaluation for the two use cases, in the BT(256) topology, where we consider both the uniform and the power-law load distributions, and the results are normalized to the all-red scenario. Not surprisingly, the network utilization of both use cases is *independent* of the specific reduce task being performed, as can be seen in Fig. 9a. This is due to the fact that our model, and in turn, SOAR, do not distinguish between the concrete details of the use case, and considers all messages as equal. As could be expected, the load distribution does bear an effect of the utilization, where the performance of SOAR improves as the distribution is more skewed (as is the case for the power-law distribution). This is attributed to the fact that for highly asymmetric load distributions, SOAR identifies the key points with severe load, and places blue nodes at (or close to) such points. On the other extreme, as the load is more evenly distributed, the judicious choices made by the algorithm have a lesser effect on overall utilization.

Fig. 9b presents the normalized cost reduction in terms of the byte complexity, where normalization is again done compared to the all-red scenario. We see that both the load distribution, and the actual application use case, affect the performance. The byte complexity in the PS use case is very similar to the utilization. This is due to the fact that we are using a non-negligible dropout rate of 0.5, as is mostly advised in distributed ML. For this case the sizes of messages traversing distinct links in the network do not vary significantly, and message sizes increase very mildly as we approach the root of the network. For the WC use case, the effect of increasing message sizes is more pronounced (as also discussed in Sec. 5.1), leading to diminished improvement in terms of byte complexity, when compared to the utilization. However, the general trends across distributions are still apparent.

Lastly, Fig. 9c shows the effect of having more blue nodes, when compared with the all-blue solution. These results highlight the effect of the message sizes on byte complexity, where for the WC use case the performance of SOAR comes very close to that of the all-blue solution, already when using but a few blue nodes. In contrast, for the PS use case the byte complexity is very closely related to the utilization complexity (as message sizes do not vary significantly). This is manifested by the fact that significantly more blue nodes are required in order to come close to the performance of the all-blue scenario. As distributed ML environments become ubiquitous, we believe that our proposed algorithm for data aggregation within the network can have a significant impact on the performance of such systems. Overall, our results indicate that although message sizes do affect the byte complexity beyond the effects manifested by the network utilization, the ability to determine the optimal location of a bounded number of blue nodes, as done by our algorithm, indeed results in performance that quickly comes close to that obtained by an unbounded solution.

Takeaways: Minimizing the network utilization reduces significantly also the byte count. The effect of using in-network processing can differ across different applications (e.g., WC and PS).

[4]We used dropout in order to obtain more diverse network utilization results in terms of bytes, as using all features would render the utilization complexity, and the number of bytes sent, the same.

[5]We note that our work considers solely the network load produced by such tasks, and not the quality of the model produced, which may depend on a variety of problem characteristics. We therefore do not implement the actual neural network, but rather consider the messages sent by the worker servers, and the aggregation of these messages.