LESS: A Matrix Split and Balance Algorithm for Parallel Circuit (Optical) or Hybrid Data Center Switching and More

Liang Liu
Georgia Institute of Technology
lliu315@gatech.edu

Jun (Jim) Xu
Georgia Institute of Technology
jx@cc.gatech.edu

Mohit Singh
Georgia Institute of Technology
mohit.singh@isye.gatech.edu

ABSTRACT

The research problem of how to use a high-speed circuit switch, typically an optical switch, to most effectively boost the switching capacity of a datacenter network, has been extensively studied. In this work, we focus on a different but related research problem that arises when multiple (say s) parallel circuit switches are used: How to best split a switching workload D into sub-workloads $D_1, D_2, ..., D_s$, and give them to the s switches as their respective workloads, so that the overall makespan of the parallel switching system is minimized? Computing such an optimal split is unfortunately NP-hard, since the circuit/optical switch incurs a nontrivial reconfiguration delay when the switch configuration has to change.

In this work, we formulate a weaker form of this problem: How to minimize the total number of nonzero entries in $D_1, D_2, ..., D_s$ (so that the overall reconfiguration cost can be kept low), under the constraint that every row or column sum of D (which corresponds to the workload imposed on a sending or receiving rack respectively) is evenly split? Although this weaker problem is still NP-hard, we are able to design LESS, an approximation algorithm that has a low approximation ratio of only $1 + \epsilon$ in practice and a low computational complexity of only $O(m^2)$, where $m = \|D\|_0$ is the number of nonzero entries in D. Our simulation studies show that LESS results in excellent overall makespan performances under realistic datacenter traffic workloads and parameter settings.

CCS CONCEPTS

• **Networks** → *Network resources allocation*; *Network performance analysis*; Data center networks.

KEYWORDS

Optical (Hybrid) Switching in Data Center Networks, Parallel Optical Switching, Matrix Split and Balance

ACM Reference Format:
Liang Liu, Jun (Jim) Xu, and Mohit Singh. 2019. LESS: A Matrix Split and Balance Algorithm for Parallel Circuit (Optical) or Hybrid Data Center Switching and More. In *IEEE/ACM 12th International Conference on Utility and Cloud Computing (UCC'19), December 2–5, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3344341.3368807

UCC '19, December 2–5, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6894-0/19/12…$15.00
https://doi.org/10.1145/3344341.3368807

1 INTRODUCTION

Data center network continues to grow both in size, as measured by the number of server racks, and in link speeds, thanks to the phenomenal growth of cloud computing services. This in turn has led to an explosive growth in the amount of traffic the data center has to switch between its server racks [5]. A cost-effective solution approach to this scalability problem, called hybrid circuit (optical) and packet switching or hybrid switching in short, has received considerable research attention in recent years [8, 17, 24, 26, 28, 29]. In a hybrid-switched data center, shown in Figure 1, n racks of computers on the left, called *input ports*, are connected by both a circuit switch and a packet switch to n racks on the right, called *output ports*. The circuit switch has a much higher bandwidth than the packet switch, but incurs a nontrivial reconfiguration delay δ when the switch *configuration* has to change. Since such a circuit switch is almost invariably an optical switch nowadays, we will always refer to a circuit switch as an optical switch in the sequel.

All existing research work on optical or hybrid switching are focused on the following optimization problem: Given a traffic demand matrix D from input ports to output ports, how to schedule the optical switch to best (e.g., in the shortest possible makespan) meet the demand? A (workable) schedule for the optical switch typically consists of a sequence of configurations (matchings) and their time durations $(P_1, \alpha_1), (P_2, \alpha_2), \cdots, (P_K, \alpha_K)$ that allow the optical switch to remove (i.e., transmit) most of the traffic demand from D, so that the remaining traffic demand is small enough for the packet switch to handle. Now, for the purpose only of simplifying our presentation, we ignore the (existence of) packet switch and formulate the hybrid switching problem as an optical switching (only) problem. We will show in §2.3 that this simplification does not change the nature and the difficulty of our research problem, to be introduced next.

1.1 Load Balance over Multiple Switches

In this work, we focus on a different but related research problem that is also orthogonal to this optical switching problem. More specifically, we formulate and solve a new load-balancing problem that naturally arises when the n racks shown in Figure 1 are connected by not just one but multiple *independent* (i.e., parallel) optical switches of the same size and capacity. Such parallel switching networks have already been proposed for boosting the total switching capacity of an optical data center network. For example, in Microsoft's ProjecToR [11], each rack is equipped with multiple (say s) independent optical transmitters and receivers, and any transmitter at a rack can pair with any (available) receiver at another rack.

All optical switching algorithms [3, 16, 18–21, 25] have computational complexities that grow at least quadratically (i.e., $O(n^2)$)

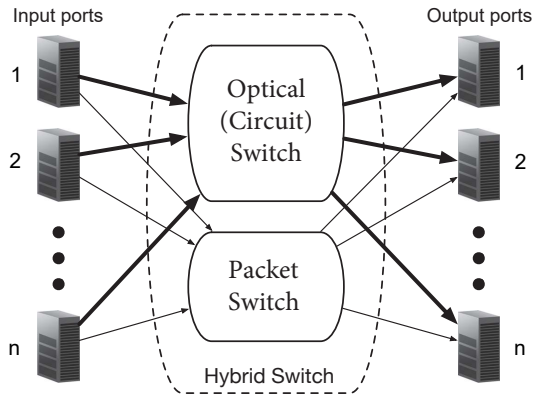

Figure 1: Hybrid Circuit and Packet Switch

with the switch size n. As will be explained in §2.1, s parallel optical switches can be naturally viewed as a giant $sn \times sn$ switch, so centrally scheduling such a giant switch using any such algorithm has a high computational complexity of at least $O(s^2n^2)$. Therefore, this scheduling problem naturally calls for the following divide-and-conquer approach: Given a traffic demand matrix D, we split D into s sub-workload matrices $D_1, D_2, ..., D_s$ and give them to the s switches as their respective workloads. This approach makes perfect systems sense, as the scheduling of each D_k on switch k can be computed independent of each other (using a different processor). We will show in §4.5 that such parallelization is critical in keeping the amount of time needed to compute all the s schedules at acceptable levels when the switch size (i.e., number of racks) n is large.

We focus on a Matrix Split and Balance (MSB) problem that lies at the heart of this divide-and-conquer approach: How to split the traffic demand D so that the resulting sub-workload matrices $D_1, D_2, ..., D_s$ lead to near-optimal switching performance yet the computation of this split is not NP-hard or otherwise extremely expensive? One intuitive notion of optimality is to minimize the worst-case (i.e., longest) makespan of the s schedules resulting from $D_1, D_2, ..., D_s$ respectively. This optimality notion is however computationally infeasible since even to minimize the makespan of a single schedule is usually NP-hard [16]. Hence, we instead impose the following two milder conditions on this split that can work *toward this optimality*.

The *first condition* is that the total traffic demand in every row of D, which corresponds to that originating at an input port, should be evenly split among $D_1, D_2, ..., D_s$, and so should every column of D, which corresponds to that destined for an output port; we call this condition *line-even* as each row or column is a straight line through the matrix. The rationale for such a split is that, since every optical switch receives roughly the same amount of workload to its $2n$ input and output ports, these s switches hopefully can finish their respective workloads in similar amounts of time (i.e., similar makespans), leading to a short *overall makespan* (the maximum among the s makespans).

Although letting $D_1 = D_2 = \cdots = D_s = D/s$ trivially satisfies this condition, this naive split is far from ideal because every nonzero entry in D is cut into s identical pieces, each of which incurs a nontrivial reconfiguration delay δ. Hence we impose the *second condition* that $\sum_{k=1}^{s} \|D_k\|_0$, the total number of nonzero entries in these s matrices, is to be minimized, which we call the *sparsity* condition. Here $\|M\|_0$ denotes the number of nonzero entries in a matrix M.

1.2 LESS: Our MSB Solution

To summarize, our MSB problem is to split D into $D_1, D_2, ..., D_s$ under the *constraint* that every row or column sum of D is evenly split and with the objective of *minimizing* the total number of nonzero entries in $D_1, D_2, ..., D_s$. Unfortunately, this relatively easier constrained minimization problem is still NP-hard [1]. The first contribution of this work is LESS (Line-Even Sparse Split), an approximation algorithm that provides the following strong theoretical guarantee: In any solution (split) produced by the LESS algorithm, the number of nonzero entries, starting with $\|D\|_0$ before the split, will not be increased by more than $(s-1)(2n-1)$ after the split. In other words, we have

$$\sum_{k=1}^{s} \|D_k\|_0 \leq \|D\|_0 + (s-1)(2n-1) \tag{1}$$

For example, suppose $s = 8$ (switches), $n = 100$ (racks), and an 100×100 traffic demand matrix D has $4,000$ nonzero entries (i.e., $\|D\|_0 = 4,000$). Then after the split, $\sum_{k=1}^{s} \|D_k\|_0 \leq 4,000 + (8-1) * (200-1) = 5,393$, and each D_k contains on average about 674 nonzero entries. In comparison, the aforementioned simple even split of D results in each D_k containing $4,000$ nonzero entries.

Although LESS guarantees a strong upper bound on the total number of nonzero entries $\sum_{k=1}^{s} \|D_k\|_0$, it does not in theory guarantee that this total is roughly evenly distributed among $D_1, D_2, ..., D_s$. However, empirically in almost all splits produced by our algorithm, $\|D_1\|_0$, $\|D_2\|_0$, ..., $\|D_s\|_0$ do have similar values, as will be shown in §4.4. Intuitively, this outcome is to be expected because, as we will explain in §3.4.2, LESS is "equal-opportunity" in the sense its logic is inherently not biased toward or against any matrix D_k in "doling out" nonzero entries. This said, it appears extremely hard, if not impossible, to prove any such evenness (of split) guarantee for this algorithm. On the other hand, if we were to modify LESS to optimize also this evenness, we would almost certainly get back into the NP-hard territory.

The LESS algorithm, based on linear programming (LP), is conceptually straightforward, as we will explain in §3.2. However, its execution time is a bit too long, since there are $O(m)$ linear equations to be satisfied by sm variables, where $m = \|D\|_0$ is the number of nonzero entries in D and is usually much larger than n. For example, when $n = 100$ and $s = 8$ (switches), it takes hundreds of milliseconds to compute a single 8-way split. In comparison, the length of a scheduling epoch is typically a few milliseconds.

Our second contribution is to reduce this execution time by converting this LP computation problem to a graph computation problem that is much less expensive. In theory the computational complexity of this graph algorithm is $O(m^2)$, but in practice the actual complexity is close to $O(m^{1.5})$, as we will show in §3.4.4. Our experiments show that the graph algorithm runs roughly an order

of magnitude faster than the LP-based algorithm when n is quite large (e.g., $n = 100$).

1.3 Beyond Optical/Hybrid Switching

A third contribution of this work is the formulation itself of the MSB (matrix split and balance) problem under the line-even and the sparsity conditions. This is a well-defined load-balancing problem that we expect will find many new applications beyond optical and hybrid switching. Furthermore, this problem is intricately connected to (to be shown in §3.1) and algorithmically equivalent to the following theoretical question that is interesting in its own right: How to split a doubly stochastic matrix into s scaled (by a factor of $1/s$) doubly stochastic matrices so that the total number of nonzero entries in the s resulting matrices is minimized? To the best of our knowledge, there had been no study of anything equivalent or similar to this question, whereas several other theoretical questions concerning doubly stochastic matrix and its Birkhoff von Neumann decomposition have received considerable research attention (e.g., [4, 7, 14]).

The rest of the paper is organized as follows. In §2, we provide a succinct description of the background and the related work. In §3, we formulate the MSB problem and present our solution LESS. Finally, we evaluate the performance of LESS in §4 and conclude the paper in §5.

2 BACKGROUND AND RELATED WORK

2.1 Parallel Hybrid/Optical Switching

As explained earlier, in studying our MSB problem, we simplify the hybrid switching problem by ignoring the (existence of the much slower) packet switch, and formulate it as an optical switching (only) problem. This simplification is justified for two reasons. First, all existing hybrid switching algorithms generate an optical switch schedule by iteratively finding and subtracting a configuration (P_k, α_k), until the remaining traffic demand is small enough for the packet switch to handle. Hence none of them would behave differently (e.g., adopt a different scheduling strategy) if "told" in advance that the capacity of the packet switch is 0. Second, every optical switching (only) algorithm is readily convertible to a hybrid switching algorithm by wrapping up the computation of the optical switch schedule whenever this termination condition is met.

Datacenter switching with multiple (say s) optical transmitters and receivers attached to each of the n racks, such as the setting in ProjecToR [11], can be first formulated as scheduling an $sn \times sn$ optical switch as follows. We are given an $n \times n$ traffic demand matrix D, and each matrix entry $D(i, j)$ is the amount of traffic that originates at Rack i and is destined for Rack j, within a short (e.g., 3 milliseconds long) scheduling epoch of the recent past (e.g., from 4 milliseconds ago to 1 millisecond ago). Our optical switching algorithm needs to meet this demand in the next scheduling epoch. We assume full knowledge of the precise and complete demand matrix D (in this recent past epoch), as do almost all prior works on hybrid switching and on optical switching. At any time t, each rack (as an input port) can transmit data to up to s other racks (output ports) and each rack (as an output port) can receive data from up to s other racks (input ports) in parallel. Furthermore, more than one (say $s'(\leq s)$) transmitters at an input port i can be paired with s' receivers at an output port j at the same time. In this case, the total transmission rate from rack i to rack j (at time t) is s' times the single (transmitter-to-receiver) link rate, which is normalized to 1 throughout the paper. The scheduling problem for the $sn \times sn$ optical switch is how to match the sn transmitters with the sn receivers over time to transmit the traffic demand D, so that the makespan of the resulting schedule is minimized.

In formulating the MSB problem in this work, we convert this $sn \times sn$ optical switching problem to that of scheduling s independent $n \times n$ virtual switches, by imposing the following slight restriction. We number the s transmitters at each input port $1, 2, \ldots, s$, and do the same to the s receivers at each output port; the number-1 transmitters at all input ports and the number-1 receivers all output ports form the first $n \times n$ virtual switch, the number-2 transmitters and the number-2 receivers form the second $n \times n$ virtual switch, and so on. In other words, the restriction is that only transmitters and receivers of the same ID (e.g., number-1, number-2, etc.) can pair with each other. This restriction is effectively almost innocuous in the following sense: There is no scheduling algorithm known to us at the moment that allows the original $sn \times sn$ switching system to deliver a significantly higher throughput than our divide-and-conquer approach, using the same or less amount of computation.

2.2 BvND of Doubly Stochastic Matrices

In this section, we explain the subtle connection between the line-even condition (that every row or column sum is evenly split among the sub-workload matrices) imposed on the MSB problem and the Birkhoff-von Neumann Decomposition (BvND) of doubly stochastic matrices, in the context of optical switching. We say that a nonnegative $n \times n$ matrix M is *doubly stochastic* (or doubly sub-stochastic) if every row or column sum of M is equal to 1 (or no larger than 1). The Birkhoff-von Neumann Theorem [2] states that a doubly stochastic (or doubly sub-stochastic) matrix M can be expressed as (or dominated by) a linear combination of permutation matrices. More precisely, we have $M = \sum_{k=1}^{K} \alpha_k P_k$ (or $M \leq \sum_{k=1}^{K} \alpha_k P_k$), where $\sum_{k=1}^{K} \alpha_k = 1$ and $P_1, P_2, \ldots, P_K$ are *permutation matrices*, in which each row or column has exactly one non-zero entry with value 1. Let M be doubly stochastic. We call any uM, where $u > 0$ is a scaling factor, a scaled doubly stochastic matrix, or u-scaled if the scaling factor u is to be emphasized. Clearly, any u-scaled doubly stochastic matrix can also be expressed as a linear combination of permutation matrices, in which the sum of the linear coefficients $\sum_{k=1}^{K} \alpha_k$ is equal to u instead of 1.

We assume there is only one optical switch for the moment. For ease of presentation, we usually normalize the following quantities to 1: the rate of the transmission link in the optical switch (between an input port and an output port), the length of a scheduling epoch, and hence the maximum amount of traffic that the link can carry during the epoch. Suppose we do that, and normalize all the elements in a traffic demand matrix D accordingly. Then, when the reconfiguration delay δ of the optical switch is assumed to be 0, by the Birkhoff-von Neumann Theorem above, we can find a schedule according to which the circuit switch can finish transmitting D within the epoch, if and only if D is doubly sub-stochastic.

We now assume there are s parallel optical switches and the normalized rate of each transmission link in each switch is $1/s$ (so that

the s links from any input port or to any output port have a combined normalized rate of 1). Again we assume the reconfiguration delay δ of every optical switch is 0. Then a doubly-stochastic traffic demand matrix D for these s switches combined is scheduleable (i.e., can be completely transmitted with a scheduling epoch), as just explained. However, if we split D into sub-workload matrices $D_1, D_2, \cdots, D_s$, every sub-workload matrix is scheduleable by the corresponding switch if and only if they are all $1/s$-scaled doubly-stochastic, which is precisely the line-even condition.

2.3 Optical/Hybrid Switching Algorithms

As explained earlier, our divide-and-conquer approach to scheduling the s parallel optical switches consists of two steps that correspond to two separate problems respectively. The first step is to split the demand matrix D into s line-even and sparse (to the extent possible) sub-workload matrices $D_1, D_2, ..., D_s$ that are fed into the s optical switches as their respective workloads. The corresponding MSB problem is the focus of this paper, The second step is for each optical switch to schedule its respective sub-workload. The corresponding problem, known as optical switching with a nontrivial reconfigurable cost, has been thoroughly studied in the optical and the hybrid (circuit and packet) switching literatures (e.g., [3, 9, 16, 18–21, 25, 27]) and is hence not a focus of this paper. Therefore, we will describe only those optical/hybrid switching concepts and algorithms that are involved in evaluating the efficacies of our LESS algorithm, such as partial reconfigurability and the Best First Fit algorithm [19].

As mentioned in §1, an $sn \times sn$ switching fabric resulting from deploying s transmitters and receivers at each rack/port has been used in Microsoft's ProjecToR [11] datacenter network. Its scheduling algorithm, briefly described and not considered as a major contribution in [11], is to schedule the $sn \times sn$ switch as a whole, so it is very different than our divide-and-conquer approach. In [11], most of the transmitters and receivers are "prewired" for handling static traffic workloads. For the remaining "free" (for matching) transmitters and receivers, their matching is modeled in [11] as a (distributed) stable marriage problem, in which a sender's preference score for a receiver is equal to the age of the data the former has to transmit to the latter in a scheduling epoch, and is solved using a variant of the Gale-Shapely algorithm [10]. Clearly, this solution is aimed at minimizing transmission latencies while avoiding starvations, and not at maximizing network throughput, or equivalently minimizing makespan. In comparison, the design objective of LESS is to minimize overall makespan to the extent possible.

In the rest of this section, we focus on how a single optical switch schedules the sub-workload (matrix) assigned to it. All existing works on optical and hybrid switching, except BFF (Best First Fit) [19] and [25], assume that the optical switch is not *partially reconfigurable* in the following sense: When the circuit switch changes from one configuration (matching) to the next, all input ports have to stop data transmission during the reconfiguration period (of duration δ), including those input ports that pair with the same output ports during both configurations.

This is however an outdated and unnecessarily restrictive assumption because nowadays optical technologies (e.g., free-space optics as used in [11, 13]) can readily support *partial reconfiguration* in the following sense: Only the input ports affected by the reconfiguration need to pay a reconfiguration delay δ, while unaffected input ports can continue to transmit data during the reconfiguration. It has been shown that this partially reconfigurable capability allows for the design of new hybrid/optical switching algorithms, such as the aforementioned BFF algorithm [19], that have much lower computational complexities, yet can deliver much better makespan (completion time) performances.

Although our MSB problem and LESS solution is mostly orthogonal to how each optical switch schedules the sub-workload (matrix) assigned to it, we assume that the optical switches have partially reconfigurable capability and use BFF [19] as their underlying scheduler, because an inefficient underlying scheduler would "muddle the water" in evaluating LESS against the naive solution (i.e., result in poor makespan performances for both).

Here we give only a brief description of BFF (Best First Fit) [19]. At the beginning of the scheduling (i.e., $t = 0$), when all input ports and all output ports are available, BFF runs a maximum weighted matching (MWM) algorithm [6] to obtain the heaviest (w.r.t. their weights in D) initial matching between inputs and outputs. Then BFF tries to match input ports with output ports as soon as they become available (i.e., after their previous transmissions are over) in the following greedy manner: Each available output port attempts to match with the best available input port (i.e., the one with the largest amount of traffic to send to the output port) at the moment, and vice versa.

3 MSB PROBLEM AND LESS SOLUTION

In this section, we first formally formulate the Matrix Split and Balance (MSB) problem in §3.1. Then in §3.2, we introduce our solution, Line-Even Sparse Split (LESS), and elaborate how to reduce LESS for a s-way MSB problem to LESS for a 2-way MSB problems. Then we describe in details how LESS solves a 2-way MSB problems by two different methods, a straightforward but slower LP-based method in §3.3, and a faster combinatorial method in §3.4.

3.1 Matrix Split and Balance (MSB)

In this section, we formally formulate the MSB problem of splitting the demand matrix D into s sub-workload matrices $D_1, D_2, ..., D_s$. It is a constrained optimization problem with the objective of minimizing the total number of nonzero entries (Formula (2)), or equivalently of maximizing the sparsity of the split. There are four sets of constraints shown in Formulae (3) through (6) respectively. In them we define $[s] \triangleq \{1, 2, \cdots, s\}$ and $[n] \triangleq \{1, 2, \cdots, n\}$.

The first set of constraints (Equations (3)) state that, for each column j in each matrix D_k, the sum of entries in the j^{th} column of D_k must be equal to $1/s$ of the sum of entries in the j^{th} column of D. These constraints ensure that the j^{th} output port of every switch is given the same amount of workload that is equal to $1/s$ of the traffic to be received by the rack j. The second set of constraints (Equations (4)) state the same for each row i in each matrix D_k. These constraints ensure that the i^{th} input port of every switch is given the same amount of workload that is equal to $1/s$ of the traffic to be transmitted by rack i. These two sets of constraints correspond to the aforementioned "line-even" (i.e., identical row

or column sum) requirement. The fourth (Inequalities (6)) and the third (Equations (5)) sets state respectively that $D_1, D_2, ..., D_s$ are *nonnegative* matrices and that their total is D.

$$\text{minimize} \quad \sum_{k=1}^{s} \|D_k\|_0 \tag{2}$$

$$\text{subject to} \quad \sum_{i=1}^{n} D_k(i,j) = \frac{1}{s}\sum_{i=1}^{n} D(i,j), \forall j \in [n], k \in [s] \tag{3}$$

$$\sum_{j=1}^{n} D_k(i,j) = \frac{1}{s}\sum_{j=1}^{n} D(i,j), \forall i \in [n], k \in [s] \tag{4}$$

$$\sum_{k=1}^{s} D_k(i,j) = D(i,j), \forall i,j \in [n] \tag{5}$$

$$0 \le D_k(i,j) \le D(i,j), \forall i,j \in [n] \tag{6}$$

Although all the constraints are linear, the objective function $\sum_{k=1}^{s} \|D_k\|_0$ is not, so this constrained optimization problem is not a Linear Programming (LP) problem. In fact, it has been proved to be NP-hard [1], so only heuristic or approximate solutions to it exist that run in polynomial time. Our solution LESS is a $(1 + \frac{(2n-1)(s-1)}{m})$-approximation algorithm, where $m = \|D\|_0$ is the number of nonzero entries in D. In practice, it can be considered a $(1+\epsilon)$-approximation algorithm, since typically $n = o(m)$, s is a small constant (typically < 10), and n can grow to hundreds (of racks) in real-world datacenter networks.

3.2 Line-Even Sparse Split (LESS)

The design of LESS is based on the following insight: The linear constraints (3) through (6) define a polytope within which any point satisfies the line-even condition and any extreme point of this polytope corresponds to a fairly sparse split in the sense of (1), which we will prove shortly. Hence our LESS algorithm is simply to find an extreme point of this polytope. This can be done by replacing the nonlinear objective function in (2) by a dummy linear objective function such as "Minimizing 0" in the constrained optimization problem above, and solving the resulting LP problem using an LP solver such as Gurobi [12].

3.2.1 Reduction from s-way Split to 2-way Splits. Throughout this section, whenever we use the term *split*, we mean to split (the matrix) in the LESS manner. In other words, such a split always corresponds to an extreme point of the corresponding polytope. Now we show that, for any $s > 2$, we can reduce an s-way split (i.e., D into $D_1, D_2, ..., D_s$) to a "binary tree" of $s-1$ recursive 2-way splits. This reduction will not only significantly simplify our presentation of the resulting linear programming (LP) problem and solution, but also allow for the use of parallel processing (to be elaborated next) to speedup its computation. Intuitively, this reduction is straightforward when s is a power of 2. For example, when $s = 8$, D is split first into "two halves", then into "four quarters", and finally into eight sub-workload matrices $D_1, D_2, ..., D_8$, each of which accounts for exactly $1/8$ of the total workload contained in D. The corresponding "binary tree" is a complete binary tree of height 3 with a total of $s - 1 = 7$ internal nodes, each of which corresponds to a 2-way split.

We now explain how this reduction can be done when s is not a power of 2. Due to the recursive nature of the splits, we need only to explain what the very first 2-way split of a matrix D' should be, when D' needs to *eventually* be split into s' pieces. There are only two cases to consider. In the first case where s' is an even number, the 2-way split has weights $(1/2, 1/2)$ in the sense each row or column sum of D'_1 is equal to $1/2$ of the corresponding row or column sum of D'. Matrices D'_1 and D'_2 will then be split further into $s'/2$ pieces each. In the second case when s' is an odd number, the 2-way spit has weights $(\frac{s'-1}{2s'}, \frac{s'+1}{2s'})$ in the sense each row or column sum of D'_1 is equal to $\frac{s'-1}{2s'}$ of the corresponding row or column sum of D'. Matrices D'_1 and D'_2 will then be split further into $(s'-1)/2$ and $(s'+1)/2$ pieces respectively. For example, when $s' = 7$, the 2-way split has weights $(3/7, 4/7)$. The resulting D'_1 and D'_2 need to be split further into 3 and 4 pieces respectively.

Now that any s-way split can be reduced to $s-1$ 2-way splits, we will only describe how a 2-way split is performed in the sequel. Furthermore, we will only consider weights $(1/2, 1/2)$ because the 2-way LESS algorithm works with any (pair of) weights (as parameters) in the same manner. When describing the 2-way LESS algorithm with weights $(1/2, 1/2)$ in the next section, we will also prove that each 2-way split increases the number of nonzero entries $\sum_{k=1}^{s} \|D_k\|_0$ by at most $2n-1$. This implies that any s-way split increases $\sum_{k=1}^{s} \|D_k\|_0$ by at most $(s-1)(2n-1)$ (Inequality (1)), since it can be reduced to $s-1$ 2-way splits.

3.2.2 Parallelization. As explained earlier, this reduction from s-way split to *2*-way splits allows for the speedup of its computation using parallelization. We now illustrate how to parallelize the computation in the aforementioned simple case of $s = 8$, where there are seven instances of 2-way split computations over three rounds: split D first into "two halves" (one instance) in the first round , then into "four quarters" (two instances) in the second round, and finally into eight sub-workload matrices $D_1, D_2, ..., D_8$ (four instances) in the third round. Clearly, four (more generally $s/2$) parallel processors (or cores) can compute this 8-way split in three rounds of time, that is, roughly three (more generally $\log_2 s$) times the amount of time needed to compute a 2-way split instance. In comparison, serial execution takes roughly seven (more generally $s-1$) rounds of time. Finally, we do not advocate further pipelining these three (more generally $\log_2 s$) rounds of computations because although it increases the "throughput" of this computation, it does not reduce the "delay", which is what matters in real-world operations.

3.2.3 Comparison with Naive Solution. Although the naive MSB solution of splitting D evenly (i.e., $D_1 = D_2 = \cdots = D_s = D/s$) satisfies all the constraints (3) through (6), it maximizes, rather than minimizes, the objective function $\sum_{k=1}^{s} \|D_k\|_0$. This leads to much higher reconfiguration costs for the naive solution, as we will show in §4. As a result, LESS outperforms the naive solutions under most of the realistic parameter settings.

3.3 LP-based *2*-way LESS

In this section, we describe the 2-way split of a matrix D' into two matrices D'_1 and D'_2 with weights $(1/2, 1/2)$. For convenience of presentation, we drop the apostrophe character from D', D'_1, and D'_2 and write them as D, D_1, and D_2 respectively. We emphasize

this (new) D could be the original demand matrix or any internal node of the aforementioned "binary tree" of 2-way splits.

This 2-way split corresponds to finding an extreme point of the polytope defined by the following equations and inequalities using the LESS algorithm. Here (7), (8), and (9) correspond to the special case of (3), (4), (5), and (6) when s is set to 2. And the weight for D_1 is the term $\frac{1}{2}$ in (7) and (8). This term will have a different value if D_1 has a different weight (e.g., 3/7 in the "odd split" example in §3.2.1). Note that we only need to compute D_1, since $D_2 = D - D_1$.

$$\sum_{j=1}^{n} D_1(i,j) = \frac{1}{2}\sum_{j=1}^{n} D(i,j), \forall i \in [n] \tag{7}$$

$$\sum_{i=1}^{n} D_1(i,j) = \frac{1}{2}\sum_{i=1}^{n} D(i,j), \forall j \in [n] \tag{8}$$

$$0 \le D_1(i,j) \le D(i,j), \forall i,j \in [n] \tag{9}$$

Note that (7) corresponds to n equations (also called *tight constraints* below in Lemma 1), one for each row i, and so does (8). Out of these $2n$ tight constraints, only $2n-1$ of them are linearly independent, because the sum of n row sums of D_1 has to be equal to the sum of n column sums of D_1. According to Lemma 1 below, any *extreme point solution* (defined precisely below in Definition 1) of this LP problem has at most $2n-1$ *variables*. In this context, a *variable* corresponds to a matrix entry $D_1(i,j)$ that is not on the boundary of (9), or in other words $0 < D_1(i,j) < D(i,j)$. Clearly, each such variable $D_1(i,j)$ increases the number of nonzero entries from one (namely $D(i,j)$) before the split to two (namely $D_1(i,j)$ and $D_2(i,j)$) after the split. This proves the following proposition.

Proposition 1. *A 2-way split of D under constraints (7) through (9) increases the total number of nonzero entries by at most $2n-1$.*

Lemma 1 (Rank Lemma, Lemma 1.2.3 in [15]**).** *Let $P = \{x : Ax \ge b, x \ge 0\}$, and let x be an extreme point solution of P such that $x_i > 0$ for each i. Then any maximal number of linearly independent tight constraints of the form $A_i x = b_i$ for some row i of A equals the number of variables.*

Definition 1 (Definition 1.2.1 in [15])**.** *Let $P = \{x : Ax \ge b, x \ge 0\} \subseteq \mathbb{R}^n$. Then $x \in P$ is an extreme point solution of P if there does not exist a nonzero vector $y \in \mathbb{R}^n$ such that $x + y, x - y \in P$.*

Such an extreme point solution can be computed using a LP solver such as Gurobi [12]. However, when n is large, this LP computation is very slow. For example, when $n = 100$ (racks), it takes the Gurobi, which is the quickest among LP solvers by our experience, hundreds of milliseconds to compute a 2-way split of D. In the next section, we describe a non-LP-based LESS algorithm that performs the same LP computation, but in a combinatorial manner. For $n = 100$, it runs an order of magnitude faster than LP-based LESS, as we will show in §4.5.

3.4 Combinatorial *2*-way LESS

The combinatorial LESS algorithm performs the same LP (solving) operation as before: Starting with the aforementioned naive solution of $D_1 = D/2$, the algorithm iteratively modifies D_1 within the solution space to push it towards one of its extremal points. However, it does so by modeling D_1 as a bipartite graph and converting this LP (solving) operation into a graph computation problem.

Algorithm 1: The pseudocode of combinatorial 2-way LESS

Input : D;
Output: D_1;
1 Initialize $D_1(i,j) \leftarrow D(i,j)/2, \forall i,j \in [n]$;
2 Convert D_1 to G;
3 **while** *An alternating cycle σ is found in G* **do**
4 Increase and decrease the weights of edges in σ in an alternating manner by the same value η so that all edge weights remain within their "legal ranges" and one or more edges become tight;
5 Remove tight edges from G;
6 **end**
7 Return D_1 that is converted back from G;

3.4.1 Conversion to Graph Computation. To describe this conversion, we need the following definition.

Definition 2. *We call a matrix entry $D_1(i,j)$ tight if $D_1(i,j) = 0$ or $D_1(i,j) = D(i,j)$. In other words, $D_1(i,j)$ is tight if it is on the boundary of the constraint $0 \le D_1(i,j) \le D(i,j)$ (as a part of (9)). We call $D_1(i,j)$ loose otherwise (i.e., when $0 < D_1(i,j) < D(i,j)$).*

In the combinatorial LESS algorithm, the matrix D_1 is modeled as a bipartite graph $G(U \cup V, E)$ whose edge set E evolves when the values of its entries are changed by the execution of the algorithm. In this bipartite graph, one partite (vertex set) U contains n vertices $u_1, u_2, \cdots, u_n$, in which each u_i, $1 \le i \le n$, corresponds to row i of D_1. The other partite V also contains n vertices $v_1, v_2, \cdots, v_n$ in which each v_j, $1 \le j \le n$, corresponds to column j of D_1. A weighted edge exists between u_i and v_j, or in other words $(u_i, v_j) \in E$, if and only if $D_1(i,j)$ is loose. The weight of this edge is set to $D_1(i,j)$.

3.4.2 Pseudocode of Combinatorial 2-way LESS. The pseudocode of the combinatorial (graph) algorithm is shown in Algorithm 1. The design of the algorithm is based on the following fact: If the bipartite graph G contains a cycle σ (Line 3), then we can modify the weights of the matrix entries (of D_1) that correspond to the edges in σ so that one or more such matrix entries become tight (Line 4). Once such a matrix entry becomes tight, its corresponding edge is removed from the bipartite graph (Line 5), according to the definition of the edge set E above. In Line 3 of Algorithm 1, the *depth-first search (DFS)* procedure is used to find a cycle.

Algorithm 1 terminates only when no cycle exists in the bipartite graph. The resulting cycle-free graph, which has only $2n$ vertices, can have no more than $2n-1$ edges (loose entries), since otherwise it cannot be cycle-free. Since each loose entry increases the number of nonzero entries by 1 as explained earlier, each 2-way split increases this number by at most $2n-1$. Hence this combinatorial view offers another proof of Proposition 1.

Now we explain why and how we can make one or more edges (rather the corresponding matrix entries) tight in each such cycle σ,

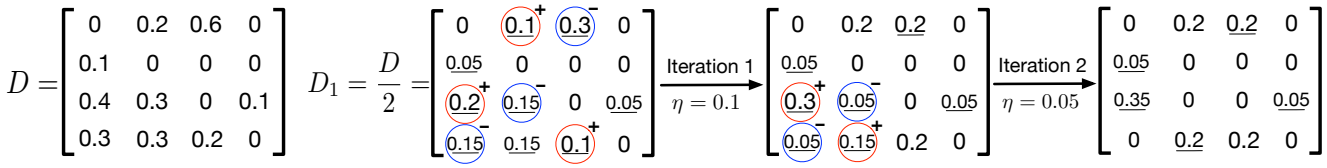

Figure 2: Example of cycle cancellation

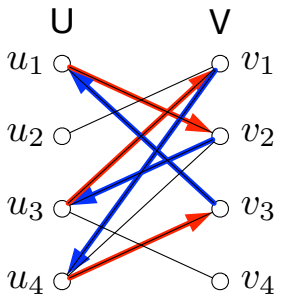

Figure 3: Alternating cycle mapping

as stated in Line 4 of Algorithm 1. Since G is bipartite, σ must contain an even number of edges. Like in the graph algorithm literature, we call σ an *alternating cycle* for a similar reason: "Walking around" the cycle starting at an arbitrary vertex on the σ and following a direction chosen arbitrarily (from the two possible directions), we will increase or decrease the weights of the edges traversed by the walk in an alternating manner, by the same amount η. In other words, we will increase the weight of the first edge by η, decrease the weight of the second edge by η, increase the weight of the third edge by η, and so on. As we will explain shortly using a toy example, this amount η is decided such that after the weight modifications, the weights of all edges (matrix entries) in σ remain within their "legal ranges" (i.e., $0 \le D_1(i,j) \le D(i,j)$ for any $D_1(i,j)$ in σ), and at least one of them becomes tight.

Since the starting point and the direction of each such walk are chosen arbitrarily, which can be implemented as being chosen randomly in practice, it appears that Algorithm 1 is "equal-opportunity" in the sense its logic is inherently not biased for or against D_1 (equivalently against or for D_2) in distributing the up to $2n-1$ new nonzero entries to D_1 and D_2. As mentioned in §1, this behavior explains why the resulting s sub-workload matrices from an s-way split have similar sparsities, as will be shown in §4.4.

3.4.3 A Toy Example. We now illustrate the concept of alternating cycle and the process of weight modification by an example shown in Figure 2 and Figure 3. The first 4×4 matrix to the left in Figure 2 is the demand matrix D, which has 9 nonzero entries. As shown in Figure 2, D_1 is initialized to $\frac{D}{2}$ (second 4×4 matrix to the left), so it also has 9 nonzero entries. All of them are loose to start with so they are all underlined. The bipartite graph corresponding to D_1 at this moment is shown in Figure 3. As explained earlier, vertices u_1, u_2, u_3, and u_4 correspond to rows 1, 2, 3, and 4 of D_1 respectively and vertices v_1, v_2, v_3, and v_4 correspond to columns 1, 2, 3, and 4 of D_1 respectively. The graph G has 9 edges, corresponding respectively to the 9 loose entries of D_1. For example, the edge $u_1 \to v_2$ corresponds to the loose (underlined) matrix entry $D_1(1,2)$.

In the first iteration, the alternating cycle $u_1 \to v_2 \to u_3 \to v_1 \to u_4 \to v_3 \to u_1$, highlighted in Figure 3 in alternating red and blue colors, is discovered. We start the cycle traversal at u_1. As specified in Line 4, we increase $D_1(1,2)$, $D_1(3,1)$, and $D_1(4,3)$, which correspond to the red edges in Figure 3 and are hence circled in red with a superscript '+' in Figure 2, and decrease $D_1(3,2)$, $D_1(4,1)$, $D_1(1,3)$, which correspond to the blue edges in Figure 3 and are hence circled in blue with a superscript '-' in Figure 2, all by the same value η.

This alternating increase and decrease by the same value has a desirable property: Any row and column sum of D_1 remains the same after the weight modifications, because an increase to a matrix entry in any row or column is always accompanied by a decrease to another entry in the same row or column, and vice versa. For example, an increase to $D_1(1,2)$ is accompanied by a decrease to $D_1(1,3)$. Due to this desirable property, the final D_1 output by Algorithm 1 satisfies (7) and (8).

We now explain how this η is determined using this example. The three matrix entries to be increased and their current values are $D_1(1,2) = 0.1$, $D_1(3,1) = 0.2$, and $D_1(4,3) = 0.1$ respectively. Their respective upper bounds are $D(1,2) = 0.2$, $D(3,1) = 0.4$, and $D(4,3) = 0.2$. The three respective differences are $D(1,2) - D_1(1,2) = 0.1$, $D(3,1) - D_1(3,1) = 0.2$, and $D(4,3) - D_1(4,3) = 0.1$. So η cannot exceed 0.1, the minimum of the three. Similarly, η cannot exceed 0.15, since the three matrix entries to be decreased are $D_1(3,2) = 0.15$, $D_1(4,1) = 0.15$, and $D_1(1,3) = 0.3$ and their lower bounds are all 0. Then η is set to the minimum of these two upper bounds, which in this case is $min\{0.1, 0.15\} = 0.1$. After these increases and decreases by $\eta = 0.1$, the new values of these matrix entries are $D_1(1,2) = 0.2$, $D_1(3,1) = 0.3$, $D_1(4,3) = 0.2$, $D_1(3,2) = 0.05$, $D_1(4,1) = 0.05$, and $D_1(1,3) = 0.2$. Among them, the values of $D_1(1,2)$ and $D_1(4,3)$ have reached their respective upper bounds (both due to an increase) $D(1,2)$ and $D(4,3)$, so both of them become tight. Hence the two corresponding edges are removed from G (so no longer underlined in the third 4×4 matrix to the left in Figure 2) after the first iteration. In Figure 3, Algorithm 1 stops after two iterations, when no more alternating cycle exists.

3.4.4 Computational Complexity of Algorithm 1. In the following analysis, we assume the $n \times n$ matrix D is not extremely sparse in the sense $n = o(m)$ where $m = \|D\|_0$ is the number of nonzero entries in D. In this case, the "while" loop in Algorithm 1 runs at most $m - (2n-1) = O(m)$ iterations because there are m edges in G to start with, each iteration removes at least one edge from G, and Algorithm 1 terminates when G contains no more than $2n-1 = o(m)$

edges. Hence, the computational complexity of Algorithm 1 is in theory $O(m^2)$ since, in each iteration, detecting a cycle using DFS has complexity $O(m)$, and so are computing η and updating edge weights. In practice, however, cycles are usually quite short for a real-world workload D, so empirically the complexity "feels more like" $O(m^{1.5})$ or less.

4 EVALUATION

In this section, we evaluate the efficacy of LESS, and compare it with that of the naive algorithm (denoted in the figures and called "Naive" in the sequel) of simply dividing D by s. We do so by feeding the sub-workload matrices resulting from LESS and Naive respectively to the s optical switches, each of which is scheduled by BFF [19]. We denote these these two resulting schedulers as LESS+BFF and Naive+BFF respectively. In this comparison, we use the overall makespan, defined as the maximum among the makespans of the schedules of the s switches, as the performance metric.

4.1 Simulation Parameters and Setup

Traffic demand matrix D: It was shown in [3, 18] that typical traffic workloads in real-world data centers exhibit two characteristics: sparsity (the vast majority of the demand matrix elements have value 0 or close to 0) and skewness (few large elements in a row or column account for the majority of the row or column sum). Hence, for our simulations, we use the same set of sparse and skewed demand matrices as used in [3, 18]. In each such matrix D, each row (or column) contains n_L large equal-valued elements (large input-output flows) that as a whole account for c_L (percentage) of the total workload to the row (or column), n_S medium equal-valued elements (medium input-output flows) that as a whole account for the rest $c_S = 1 - c_L$ (percentage), and noises. Hence n_L and n_S control the sparsity, and c_L and c_S control the skewness, of the traffic demand, respectively. Roughly speaking, we have

$$D = \sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N} \tag{10}$$

where each P_i and each P'_i is an $n \times n$ random permutation matrix.

Same as in [3, 18], in our simulation studies, the default values of the sparsity parameters n_L and n_S are set to 4 and 12 respectively and the default values of c_L and c_S are set to 0.7 (*i.e.*, 70%) and 0.3 (*i.e.*, 30%) respectively. In other words, in each row (or column) of the demand matrix, by default the 4 large flows account for 70% of its total traffic demand, and the 12 medium flows account for the rest 30%. We will also vary the sparsity parameters n_L and n_S and skewness parameters c_L and c_S in our evaluations. In Equation (10), before a noise matrix $\mathcal{N}$ (described next) is added to it, each such D is doubly stochastic (defined in §2.2). As shown in Equation (10), we also add a noise matrix term $\mathcal{N}$ to D, like in [3, 18]. Each nonzero element in $\mathcal{N}$ is a Gaussian random variable that is added to a traffic demand matrix element that was nonzero before the noise added. Each nonzero (noise) element here in $\mathcal{N}$ has a standard deviation, which is equal to 0.3% of the normalized workload 1.

Reconfiguration delay of the optical switch δ: The larger δ is, the more time the optical switch has to spend on reconfigurations, and hence the higher the resulting makespan is. By default, $\delta = 0.04$ (i.e., 4% of the scheduling epoch), although we will vary δ in our simulation studies. Here we use a larger default value of δ than that in [3, 18], which is $\delta = 0.01$ (i.e., 1% of the scheduling epoch), because the former is closer to the δ values of real-world large (e.g., 100×100) optical switches that range mostly from hundreds of μs to milliseconds [22] (after being normalized by the typical epoch length of 3 milliseconds). Although δ values as small as $12\mu s$ were mentioned in both [11] and [22], they apply only to a small switch (e.g., 4×4) or an optical transmitter-receiver pair with tiny

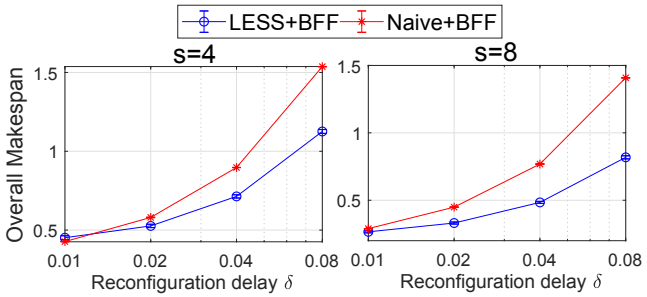

Figure 4: LESS+BFF vs. Naive+BFF while varying δ

4.2 Under Different System Parameters

In this section, we compare the overall makespan performances of LESS+BFF and Naive+BFF for different value combinations of s (number of parallel switches) and δ (reconfiguration delay), under traffic demand D with the default parameter settings described above (4 large flows and 12 small flows accounting for roughly 70% and 30% of the total traffic demand into each input port). The simulation results, presented in Figure 4, show that LESS+BFF outperforms Naive+BFF, as indicated by shorter overall makespans, when the reconfiguration delay δ is large (say $\delta \geq 0.02$). More specifically, when $\delta = 0.04$ and $s = 8$, LESS+BFF results in approximately 59% shorter overall makespan than Naive+BFF; when $\delta = 0.08$ and $s = 8$, LESS+BFF results in approximately 74% shorter overall makespan than Naive+BFF. Although error bars representing 95% confidence intervals are used in Figure 4, they are barely noticeable since, for every case (point) in the figure, the simulation results are very close to one another.

Figure 4 also shows that when δ is mall (say $\delta \leq 0.01$), LESS+BFF results in similar or slightly longer overall makespan than Naive+BFF. Our explanation is as follows. With a LESS split, the sub-workloads $D_1, D_2, ..., D_s$ are line-even but not identical matrices, and although these s matrices have superb total sparsity (i.e., $\sum_{k=1}^{s} \|D_k\|_0$), there can be some variations in their individual sparsities. Due to these variations among the sub-workload matrices and their individual sparsities, the makespans of the s switches can have some variations. In comparison, with a naive split, all s switches are given identical sub-workloads, so the resulting s makespans are identical. Since the performance metric (overall makespan) is the maximum of these s makespans, this identicalness gives the naive solution

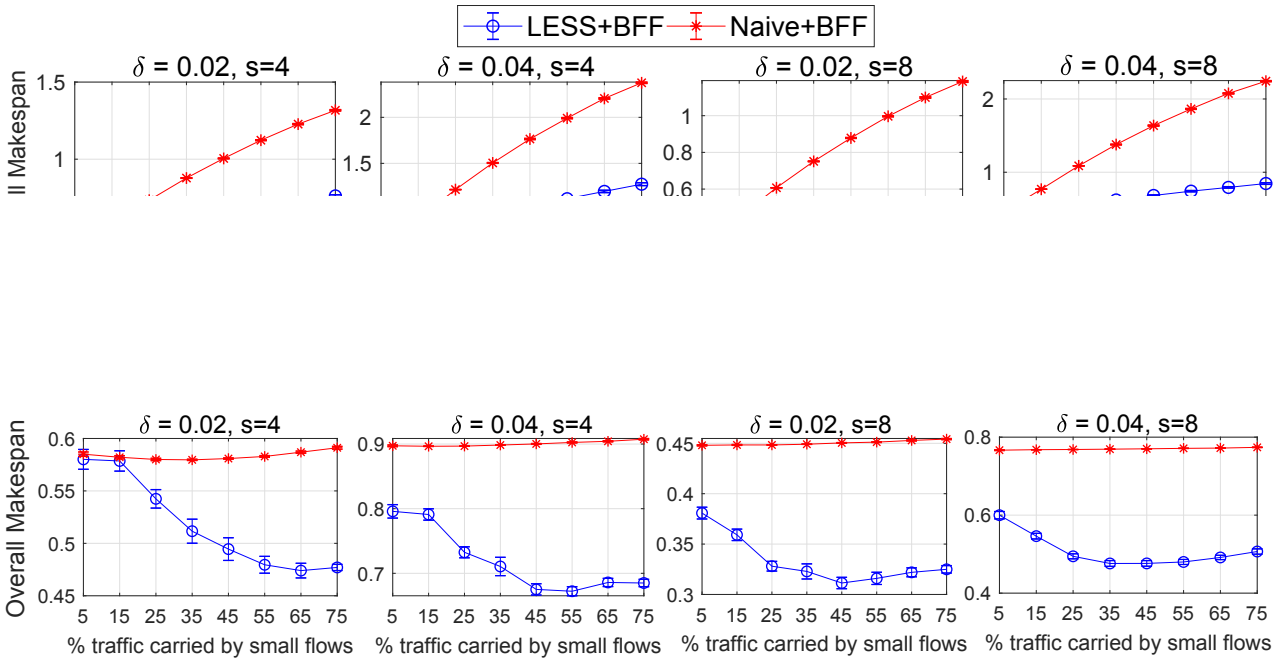

Figure 6: LESS+BFF vs. Naive+BFF while varying skewness of D

a performance edge over LESS. As a result, when δ is very small as in this case or when D is extremely sparse (e.g., in the case of $n_L + n_S = 8$ to be presented in §4.3), LESS's performance gain from the total sparsity of the split could be dwarfed by naive solution's performance edge from this identicalness.

This said, LESS+BFF may still outperform Naive+BFF in this case (of $\delta = 0.01$) when the performance metric is changed to the average makespan of all s switches. For example, when $\delta = 0.01$ and $s = 4$, although the overall makepsan for LESS+BFF (= 0.4516) is longer than that for Naive+BFF (= 0.4270), the average makepsn for LESS+BFF (= 0.4192) is actually shorter than that of Naive+BFF (= 0.4270).

4.3 Under Different Traffic Demands

In this section, we compare the overall makespan performances of LESS+BFF and Naive+BFF under a diverse set of traffic demand matrices that vary by sparsity and skewness. We control the sparsity of the traffic demand matrix D by varying the total number of flows ($n_L + n_S$) in each row from 8 to 64, while fixing the ratio of the number of large flow to that of small flows (n_L/n_S) at $1:3$. We control the skewness of D by varying c_S, the total percentage of traffic carried by small flows, from 5% (most skewed as large flows carry the rest 95%) to 75% (least skewed). In all these evaluations, we consider four different value combinations of system parameters δ and s: (1) $\delta = 0.02, s = 4$; (2) $\delta = 0.04, s = 4$; (3) $\delta = 0.02, s = 8$; and (4) $\delta = 0.04, s = 8$.

Figure 5 compares the overall makespan performances of LESS+BFF and Naive+BFF when the sparsity parameter $n_L + n_S$ varies from 8 to 64 and the value of the skewness parameter c_S is fixed at 0.3. Figure 6 compares the overall makespan performances of LESS+BFF and Naive+BFF when the skewness parameter c_S varies from 5% to 75% and the sparsity parameter $n_L + n_S$ is fixed at 16 (= 4 + 12). In each figure, the four subfigures correspond to the four value combinations of δ and s above. Both Figure 5 and Figure 6 show that LESS+BFF invariably results in shorter overall makespans than Naive+BFF, under various traffic demand matrices. In Figure 5, LESS+BFF performs consistently better than Naive+BFF, except in some cases where the traffic matrices are extremely sparse (more specifically where $n_L + n_S = 8$).

Although the reason for these outliers has been explained in the previous section, we zoom in on the case of $n_L + n_S = 8$ and $\delta = 0.02$ to emphasize that LESS is "not to blame". In this case, the average number of nonzero entries in a sub-workload matrix resulting from Naive is 731 (or 7.31 per row or column) whereas that from LESS is only 352 (or 3.52 per row or column). Hence on average, an input port pays 7.31δ reconfiguration cost in the case of Naive and 3.52δ reconfiguration cost in the case of LESS. However, when $\delta = 0.02$, this advantage of LESS in reconfiguration cost is dwarfed by the identicalness advantage enjoyed by Naive.

4.4 Sparsity Evenness of LESS

In this section, we show that, the s sub-workload matrices resulting from LESS generally have similar sparsities (numbers of nonzero entries) empirically as measured by their normalized mean absolute deviation (NMAD), although as explained earlier this property is not theoretically guaranteed. The mean absolute deviation (MAD, or average absolute deviation) of a data set $\{x_1, x_2, ..., x_n\}$ is defined as the average distance between x_i and its mean $\bar{x}$: $\frac{1}{n}\sum_{i=1}^{n} \mid x_i - \bar{x} \mid$. The normalized mean absolute deviation (NMAD)

is defined as MAD divided by $\bar{x}$. Smaller NMAD means better evenness. Table 1 shows the mean and the 95% percentile of NMAD of $\{\|D_1\|_0, \|D_2\|_0, ..., \|D_s\|_0\}$ (the number of nonzero entries in the s sub-workload matrices), for $s = 4$ and $s = 8$, under D with the default parameter settings ($n_L = 4$, $n_S = 12$, $c_L = 0.7$, $c_S = 0.3$). Table 1 shows that the average NMAD is only 5% (i.e., deviates 5% from the mean on average) when $s = 4$ and only 4% when $s = 8$.

	$s = 4$	$s = 8$
Mean NMAD	5.00%	4.01%
95%-percentile NMAD	7.36%	5.61%

Table 1: Variations among $\{\|D_1\|_0, \|D_2\|_0, ..., \|D_s\|_0\}$

4.5 Execution Times of LESS

In this section, we compare the (single-processor) execution times of LP-based LESS and combinatorial LESS, both implemented in C++, under D with the default parameter settings ($n_L = 4$, $n_S = 12$, $c_L = 0.7$, $c_S = 0.3$), on an Apple MacBook Air laptop equipped with an 1.6 GHz Intel Core i5 processor and 8 GB 2133 MHz LPDDR3. We select Gurobi [12] as the LP solver in the former algorithm due to its superior computational efficiency. As shown in Table 2, the execution times of the combinatorial LESS are roughly an order of magnitude shorter than those of LP-based LESS.

The former are already generally lower than the execution times of BFF (the underlying optical/hybrid switching algorithm), which as reported in [19] is much more computationally efficient than any other hybrid switching algorithm. With parallel processing (described in §3.2.2), the former can be further improved by 20% to 40%, as we have estimated through experiments.

This said, as mentioned earlier, the epoch duration is typically a few milliseconds long (e.g., $3ms$), so ideally the execution time of LESS should be no more than that. Currently, with software implementation, our combinatorial algorithm takes roughly an order of magnitude longer, when $n = 100$ (i.e., 100×100 switch) and $k = 8$ (parallel switches). However, we believe this execution time gap can be closed with ASIC implementation, because our combinatorial algorithm is heavy on memory I/O (mostly linked list traversals), which can be done much faster if all data reside in on-chip SRAM. The SRAM cost of ASIC implementation is quite low: Only tens of KBs of SRAM is needed when $n = 100$ and $k = 8$.

Although the focus of this work is on the MSB problem and the LESS solution, we understand that for LESS to be practically useful, its "companion" scheduler, which throughout this paper is BFF, also should have an execution time not exceeding the epoch duration. While with software implementation BFF [19] takes around $20 - 30ms$ to compute a schedule when $n = 100$, we believe it too can be sped up by an order of magnitude, by replacing the expensive maximum weighted matching (MWM) computation (at the beginning) with a much less expensive but slightly lower-quality matching computation (e.g., using iSLIP [23]) and by using the ASIC implementation.

	$s = 2$	$s = 4$	$s = 8$
Combinatorial	11.51ms	23.35ms	35.26ms
Gurobi [12]	85.72ms	216.55ms	431.02ms

Table 2: Execution Time Comparison

5 CONCLUSION

In this work, we formulate a matrix split and balance (MSB) problem that naturally arises in an optical- or hybrid-switched datacenter network where racks of servers are connected by multiple parallel optical switches. A roughly equivalent formulation of this MSB problem is "how to split a doubly stochastic matrix D into matrices $D_1, D_2, ..., D_s$ such that each of them is $1/s$-scaled doubly stochastic and the total number of nonzero entries in these s matrices is minimized?" As this problem is NP-hard, we propose LESS (Line-Even Sparse Split), an approximation algorithm that has a low approximation ratio of only $1 + \epsilon$ in practice and a low computational complexity of only $O(m^2)$, where $m = \|D\|_0$ is the number of nonzero elements in D. Our simulation studies show that LESS results in excellent overall makespan performances under realistic datacenter traffic workloads and parameter settings.

ACKNOWLEDGMENT

This work is supported in part by US NSF through award CNS-1909048, AF-1910423, and AF-1717947.

REFERENCES

[1] Edoardo Amaldi and Viggo Kann. 1998. On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. *Theoretical Computer Science* 209, 1-2 (1998), 237–260.
[2] D. Birkhoff. 1946. Tres observaciones sobre el algebra lineal. *Universidad Nacional de Tucuman Revista , Serie A* 5 (1946), 147–151.
[3] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. 2016. Costly circuits, submodular schedules and approximate carathéodory theorems. In *SIGMETRICS*. ACM, 75–88.
[4] Richard A Brualdi. 1982. Notes on the Birkhoff algorithm for doubly stochastic matrices. *Canad. Math. Bull.* 25, 2 (1982), 191–199.
[5] Casimer DeCusatis. 2014. Optical interconnect networks for data communications. *J. Lightw. Technol.* 32, 4 (2014), 544–552.
[6] Ran Duan and Hsin-Hao Su. 2012. A scaling algorithm for maximum weight matching in bipartite graphs. In *SODA*. SIAM, 1413–1424.
[7] Fanny Dufossé and Bora Uçar. 2016. Notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra Appl.* 497 (2016), 108–115.
[8] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: a hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.* 40, 4 (2010), 339–350.
[9] Shu Fu, Bin Wu, Xiaohong Jiang, Achille Pattavina, Lei Zhang, and Shizhong Xu. 2013. Cost and delay tradeoff in three-stage switch architecture for data center networks. In *HPSR*. IEEE, 56–61.
[10] David Gale and Lloyd S Shapley. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15.
[11] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. 2016. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*. 216–229.
[12] Inc Gurobi Optimization. 2019. Gurobi optimizer 8.1.1. *URL http://www.gurobi.com* (2019).
[13] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. 2014. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-space Optics. In *Proceedings of the ACM SIGCOMM*. 319–330.
[14] Janardhan Kulkarni, Euiwoong Lee, and Mohit Singh. 2017. Minimum Birkhoff-von Neumann Decomposition. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 343–354.

[15] Lap Chi Lau, Ramamoorthi Ravi, and Mohit Singh. 2011. *Iterative methods in combinatorial optimization*. Vol. 46. Cambridge University Press.

[16] Xin Li and Mounir Hamdi. 2003. On scheduling optical packet switches with reconfiguration delay. *IEEE J. Sel. Areas Commun.* 21, 7 (2003), 1156–1164.

[17] Odile Liboiron-Ladouceur, Isabella Cerutti, Pier Giorgio Raponi, Nicola Andriolli, and Piero Castoldi. 2011. Energy-efficient design of a scalable optical multiplane interconnection architecture. *IEEE Journal of Selected Topics in Quantum Electronics* 17, 2 (2011), 377–383.

[18] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. 2015. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *ACM CoNEXT (CoNEXT '15)*. ACM, New York, NY, USA, Article 41, 13 pages. https://doi.org/10.1145/2716281.2836126

[19] Liang Liu, Long Gong, Sen Yang, Jun Xu, and Lance Fortnow. 2018. Best First Fit (BFF): An Approach to Partially Reconfigurable Hybrid Circuit and Packet Switching. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 426–433.

[20] Liang Liu, Long Gong, Sen Yang, Jun Jim Xu, and Lance Fortnow. 2018. 2-Hop Eclipse: A Fast Algorithm for Bandwidth-Efficient Data Center Switching. In *International Conference on Cloud Computing*. Springer, 69–83.

[21] Liang Liu, Jun Xu, and Lance Fortnow. 2018. Quantized BvND: A Better Solution for Optical and Hybrid Switching in Data Center Networks. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 237–246.

[22] William M. Mellette, Alex C. Snoeren, and George Porter. 2018. Toward optical switching in the data center. In *High Performance Switching and Routing*. IEEE.

[23] Nick McKeown. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking* 2 (1999), 188–201.

[24] Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, and Yueping Zhang. 2010. Proteus: a topology malleable data center network. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 8.

[25] Brian Towles and William J Dally. 2003. Guaranteed scheduling for switches with configuration overhead. *IEEE/ACM Transactions on Networking* 11, 5 (2003), 835–847.

[26] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. 2010. c-Through: Part-time optics in data centers. In *SIGCOMM Comput. Commun. Rev.*, Vol. 40. ACM, 327–338.

[27] Bin Wu and Kwan L Yeung. 2006. Nxg05-6: Minimum delay scheduling in scalable hybrid electronic/optical packet switches. In *GLOBECOM*. IEEE, 1–5.

[28] Kang Xi, Yu-Hsiang Kao, and H Jonathan Chao. 2013. A petabit bufferless optical switch for data center networks. In *Optical interconnects for future data center networks*. Springer, 135–154.

[29] Xiaohui Ye, Yawei Yin, SJ Ben Yoo, Paul Mejia, Roberto Proietti, and Venkatesh Akella. 2010. DOS: A scalable optical switch for datacenters. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 24.