Scheduling Parallel Optical Circuit Switches for AI Training

Kevin Liang, Litao Qiao, Isaac Keslassy, *Senior Member, IEEE* and Bill Lin, *Senior Member, IEEE*

***Abstract*—The rapid growth of AI training has dramatically increased datacenter traffic demand and energy consumption, which has motivated renewed interest in optical circuit switches (OCSes) as a high-bandwidth, energy-efficient alternative for AI fabrics. Deploying multiple parallel OCSes is a leading alternative. However, efficiently scheduling time-varying traffic matrices across parallel optical switches with non-negligible reconfiguration delays remains an open challenge.**

We consider the problem of scheduling a single AI traffic demand matrix D over s parallel OCSes while minimizing the makespan under reconfiguration delay δ. Our algorithm SPECTRA relies on a three-step approach: Decompose D into a minimal set of weighted permutations; Schedule these permutations across parallel switches using load-aware assignment; then Equalize the imbalanced loads on the switches via controlled permutation splitting. Evaluated on realistic AI training workloads (GPT model and Qwen MoE expert routing) as well as standard benchmarks, SPECTRA vastly outperforms a baseline based on state-of-the-art algorithms, reducing schedule makespan by an average factor of $1.4\times$ on GPT AI workloads, $2.7\times$ on MoE AI workloads, and $2.4\times$ on standard benchmarks. Further, the makespans achieved by SPECTRA consistently approach newly derived lower bounds.

***Index Terms*—Optical circuit switch, matrix decomposition, AI datacenter**

I. INTRODUCTION

DATACENTER networks have had trouble scaling at the speed of large-scale AI training compute, especially lagging the AI bandwidth and power constraints [2]–[4]. Unlike traditional cloud workloads, AI training compute produces sustained high bisection bandwidth demand with repeated collective communication patterns across a large number of accelerators [5]. Due to its tight iteration-level synchronization constraints, it is extremely sensitive to the worst-case completion time, i.e., the collective makespan also known as collective completion time (CCT). These properties stress networks based on conventional electronic packet-switched fabrics, whose power and bandwidth abilities scale poorly with network size [2].

There has been a significant body of literature proposing the use of optical circuit switches (OCSes) in datacenters as a compelling alternative [2], [6]–[10]. In comparison with electronic packet switches, OCSes provide much higher bandwidth at much lower per-bit energy, both of which are important in meeting the rapidly increasing demands on AI datacenter networks. To increase aggregated capacity, prior work increasingly considers employing *parallel OCSes* [11]–[15]. On the other hand, OCSes suffer from reconfiguration delays that can significantly affect the CCT of the AI training collectives.

In this paper, we consider the problem of scheduling an AI traffic demand matrix over a set of parallel OCSes with reconfiguration delays, while minimizing the makespan (and therefore the CCT) across the parallel switches.

Related work. The existing literature mostly minimizes schedule length either assuming a single optical switch, or a multi-stage switch architecture that can be abstracted as a single switch [7]–[10]. In particular, these existing works typically focus on the scheduling of a given traffic demand matrix D onto an OCS by decomposing D into a sequence of configurations of given durations, such that the traffic demands from D are served. A critical difference with an analogous decomposition problem for electronic switches is that OCSes incur a nontrivial reconfiguration delay δ when the *switch configuration* has to change, which adds to the total delay (the *makespan*) in servicing D. Existing works (e.g., [7]–[9]) are effective in solving this scheduling problem with reconfiguration delays for the single OCS case.

LESS [11] is the most relevant work on scheduling a demand matrix D for s parallel OCSes. It splits D into s sub-matrices using a balancing criterion with a sparsity objective, then schedules each sub-matrix on a separate OCS. However, LESS focuses more on a partial-reconfiguration scheduling algorithm intended for partially-reconfigurable switches. Additional works consider parallel OCSes, but under different assumptions: They (1) achieve a distributed scheduling [12], but with lower matching efficiency; (2) use OCSes to route around failures [13]; (3) extend results to a multi-hop schedule in which nodes can serve as intermediate nodes for other flows [14], [16], [17], which we do not consider in this paper; and (4) focus on integer-weight decompositions [15].

Contributions. We introduce the SPECTRA algorithm for scheduling an AI demand matrix D across s parallel switches with reconfiguration delay, while minimizing a CCT-based makespan objective. A key idea of SPECTRA is to cut the problem into three successive sub-problems, each practically solved with polynomial-time algorithms: (i) DECOMPOSE D into a set of permutations, (ii) SCHEDULE these permutations across the s switches via load-aware scheduling, then (iii) EQUALIZE the loads on the switches by splitting the durations of some of the permutations on the most-loaded switches and moving part of their load to the least-loaded ones. We rigorously derive new lower bounds on the achievable makespan for arbitrary demand matrices by any parallel OCS scheduling algorithm. We also introduce a new Qwen-57B Mixture of Experts (MoE) workload by measuring its traffic in a 64-

K. Liang, L. Qiao and B. Lin are with UC San Diego (e-mails: {k3liang, l1qiao, billlin}@ucsd.edu). I. Keslassy is with the Technion and UC Berkeley (isaac@technion.ac.il). This work was partly supported by the Louis and Miriam Benjamin Chair in Computer-Communication Networks, NSF Award No. 2410053, and the Israel Science Foundation (grant No. 2895/26). A preliminary version of this work appeared in IEEE Networking Letters [1]. This work has been accepted for publication in the IEEE/Optica Journal of Lightwave Technology. © 2026 IEEE. Personal use permitted; republication requires IEEE permission.

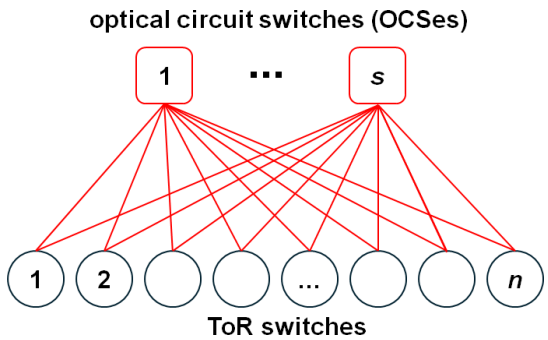


Fig. 1. Datacenter network topology with s parallel OCSes.

GPU cluster. We then compare SPECTRA against a BASELINE algorithm that extends the sparsity ideas of LESS [11], as well as against an ECLIPSE [7]-based version of SPECTRA, using three workloads: (i) a GPT-based AI training model, (ii) the new Qwen MoE measurements, and (iii) a standard benchmark in the literature. SPECTRA vastly outperforms BASELINE, with a schedule that is on average shorter by a factor of $1.4\times$ on GPT AI workloads, $2.7\times$ on MoE AI workloads, and $2.4\times$ on standard benchmarks. It also always outperforms the ECLIPSE-based variant. More importantly, the makespans achieved by SPECTRA are consistently approaching the theoretically derived lower bounds, thus achieving near-optimal performance in practice with fast polynomial-time algorithms, with robust performance across sparse and dense traffic regimes. Our results show that algorithmic co-design of traffic decomposition and parallel OCS scheduling is critical for improving optical circuit switching for next-generation AI datacenters. Our code is available at https://github.com/k3liang/SPECTRA.

II. Problem Formulation

Demand matrix. As Fig. 1 illustrates, we consider a datacenter core of s parallel *optical circuit switches* (OCSes). Each OCS has n input and output ports, connected to the Top-of-Rack (ToR) switches of n datacenter racks. Periodically, a centralized controller computes a new $n \times n$ traffic demand matrix D. Each D_{ij} corresponds to the normalized volume of data that needs to be switched from rack i to rack j over the next period. We assume that D is doubly substochastic, i.e., each row and column sums up to at most one. We call D doubly stochastic when they all sum up to exactly one.

Switch schedule. Each OCS has a schedule that consists of a sequence of permutations, where each input rack i is connected to a single output rack j. A sequence of k permutations $P_1, P_2, \ldots, P_k$, with corresponding weights $\alpha_1, \alpha_2, \ldots, \alpha_k > 0$ is said to *cover* D if the following is satisfied:

$$\sum_{i=1}^{k} \alpha_i P_i \geq D. \tag{1}$$

In addition, the optical switch needs a *reconfiguration delay* δ before each permutation. Thus, the time needed to schedule the k permutations in a single switch is

$$\sum_{i=1}^{k} (\delta + \alpha_i) = k \cdot \delta + \sum_{i=1}^{k} \alpha_i. \tag{2}$$

Makespan. We now consider the s schedules of the s optical switches. As above, consider some switch h, with $h \in [1, s]$. Assume that each such switch uses a schedule with k^h permutations P_i^h and their corresponding weights α_i^h, for $i \in \left[1, k^h\right]$. Then, extending Eq. (1), the set of all s switch schedules covers D when

$$\sum_{h=1}^{s} \left[\sum_{i=1}^{k^h} \alpha_i^h P_i^h \right] \geq D. \tag{3}$$

The *makespan* of the s parallel switch schedules is the delay of the longest one, i.e., using Eq. (2),

$$\max_{1 \leq h \leq s} \left[\sum_{i=1}^{k^h} \left(\delta + \alpha_i^h \right) \right]. \tag{4}$$

Objective. The goal of this paper is to find all permutations and weights in the s schedules of the s parallel optical switches, so that they manage to *cover* D (Eq. (3)) while *minimizing the makespan*.

The general minimization problem of the above objective is NP-hard, even with $s = 1$ [18]. Thus, in this paper, we focus on finding a practical algorithm that typically *attains a small makespan in practice*.

III. Algorithms

A. Overview

In this section, we introduce our SPECTRA algorithm (short for **S**cheduling **p**aral**l**el **c**ircuit switches for data center **tra**ffic). As formally defined in §II, SPECTRA finds s schedules of D over s parallel optical switches, such that the schedules *cover* D while attaining a *small makespan*. We start by illustrating SPECTRA's three-step approach using an example, before formally defining each step.

DECOMPOSE. The first step in SPECTRA is to DECOMPOSE a demand matrix D into a set of permutations $\{P_1, P_2, \ldots, P_k\}$ with a corresponding set of weights $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ such that their weighted sum *covers* D, i.e., $\sum_{i=1}^{k} \alpha_i P_i \geq D$.

Consider the demand D shown in Fig. 2. Fig. 3 provides an example of how D can be decomposed into 3 permutations, with the corresponding weights 0.61, 0.3, and 0.1. Fig. 3 also shows that their weighted sum indeed covers D. As detailed later in this section, given at most k nonzero elements in any row or column of D, our DECOMPOSE algorithm *guarantees* that exactly k permutations will be generated to cover D. It minimizes the number of reconfiguration delays needed while also reducing the sum duration of the permutations.

SCHEDULE. After the DECOMPOSE step, we next SCHEDULE the k permutations across the s switches. This step is equivalent to scheduling k jobs across s machines, with each α_i representing the amount of time that a job (permutation) P_i needs to run, with added consideration for a reconfiguration delay δ each time a machine (switch) needs to be (re)configured to run a job (switch permutation).

For this step, we first sort the permutations by non-increasing α_i weights, and then we schedule the corresponding P_i in non-increasing-weight-order to the *least* loaded switch

$$D = \begin{pmatrix} 0.6 & 0.3 & 0 & 0.1 \\ 0 & 0.61 & 0.39 & 0 \\ 0 & 0.09 & 0.61 & 0.3 \\ 0.4 & 0 & 0 & 0.6 \end{pmatrix}$$

Fig. 2. Example of demand matrix D.

each time. Fig. 4 illustrates the SCHEDULE process for the 3 permutations from Fig. 3 across $s = 2$ switches with reconfiguration delay $\delta = 0.01$. Initially, both switches are empty, as shown in Fig. 4. We first schedule the permutation with $\alpha_1 = 0.61$ to the first switch, so the load of the first switch becomes $0.61 + 0.01 = 0.62$. We next schedule the permutation with $\alpha_2 = 0.3$ to the second switch since it is empty and hence the least loaded. The load of the second switch becomes $0.3 + 0.01 = 0.31$. Finally, we schedule the last permutation with $\alpha_3 = 0.1$ to the second switch again, since it remains the least loaded of the two switches, making its final load 0.42. The makespan after the SCHEDULE step is $\max(0.62, 0.42) = 0.62$, as shown in Fig. 4(a).

EQUALIZE. As the final step, we aim to EQUALIZE the loads on the switches by splitting the duration (α_i) of a permutation (switch configuration) P_i into *multiple* segments and moving some segments to different switches in order to load-balance the workloads. In particular, our EQUALIZE step iteratively balances the loads between the *most* loaded switch h_{max} and the *least* loaded switch h_{min} by moving a portion of the longest-duration permutation in h_{max} to h_{min}. This is shown in Fig. 4(b), where we split $\alpha_1 = 0.61$ in the first switch into two parts, an $\alpha_1^1 = 0.515$ part that remains in the first switch and an $\alpha_1^2 = 0.095$ part that moves to the second switch, resulting in the final makespan of 0.525, as shown in Fig. 4(b).

Each of the above steps is further detailed below.

B. DECOMPOSE

Support matrix. Given an $n \times n$ matrix D, let $S \in \{0,1\}^{n \times n}$ be its *support* matrix defined by $S_{ij} = 1$ if $D_{ij} > 0$ and $S_{ij} = 0$ otherwise. Let k be the *degree* of D, corresponding to the maximum number of nonzero elements in any row or column in D. By definition, the support matrix S for D will also have a degree of k, corresponding to a maximum of k 1's in any row or column in S. The following property follows from König's Line Coloring Theorem:

Property 1. *k permutations are necessary and sufficient to cover any matrix of degree k.*

DECOMPOSE intuition. As Fig. 4(a) illustrates, the sum of all the schedule delays will include (1) δ times *the number of permutations* (in this example, $0.01 \cdot 3$), and (2) *the sum of the permutation weights* ($0.61 + 0.3 + 0.1 = 1.01$). Thus, we want to minimize both of these components.

(1) First, by Property 1, k permutations can cover D via some $\alpha_1, \alpha_2, \ldots, \alpha_k$ such that $\sum_{i=1}^{k} \alpha_i P_i \geq D$. In our approach, the goal of *minimizing the number of permutations* is to later minimize the number of reconfiguration delays.

Algorithm 1 DECOMPOSE

Input: D and δ
Assign: S = support matrix of D, $D_{rem} = D$, $S_{rem} = S$, $i = 1$, $\mathcal{A} = \{\}$ and $\mathcal{P} = \{\}$
while $S_{rem} \neq \emptyset$ **do**
  $P_i \leftarrow$ MWM-Node-Coverage(D_{rem}, S_{rem})
  $\alpha_i \leftarrow \min_{(a,b),(P_i)_{ab}=1}(D_{rem})_{ab}$
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_i\}$
  $\mathcal{A} \leftarrow \mathcal{A} \cup \{\alpha_i\}$
  $D_{rem} \leftarrow D_{rem} - (\alpha_i P_i)$
  $S_{rem} \leftarrow S_{rem} - (P_i \cap S_{rem})$
  $i \leftarrow i + 1$
$\hat{\mathcal{A}} \leftarrow$ REFINE$(D, \mathcal{A}, \mathcal{P})$
return $(\hat{\mathcal{A}}, \mathcal{P})$

Therefore, our approach is to decompose D into exactly this minimum number k of permutations.

(2) However, there may be many possible sets of k permutations that can cover D. Among these, we want to find a set of permutations along with a set of parameters $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ that also *minimizes the total duration* $\sum_{i=1}^{k} \alpha_i$.

The goal of these two properties is to avoid a decomposition with many coefficients α_i that are smaller than the reconfiguration delay δ. The usual Birkhoff decompositions [10] only target $\sum_i \alpha_i \leq 1$. However, this comes at the cost of the number of reconfigurations increasing significantly, with many small coefficients α_i, thus also increasing the total makespan due to the additional reconfiguration delay δ for each configuration. By bounding the number of reconfigurations to be exactly k, we may increase $\sum_i \alpha_i$ to slightly exceed one, but in return, reconfiguration cost and the makespan greatly decrease.

DECOMPOSE pseudocode. Alg. 1 shows the pseudocode of our DECOMPOSE algorithm. Given D, we first derive its support matrix S. DECOMPOSE then iteratively generates a set of k permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ that cover S by solving a *maximum weight matching* (MWM) problem *under node coverage constraints* [19] in each iteration.

At a high-level, we track the *remaining traffic* in D and the *remaining uncovered elements* in S after each round, denoted as D_{rem} and S_{rem}, respectively. They are initially $D_{rem} = D$ and $S_{rem} = S$. Based on the updated D_{rem} and S_{rem} in each round, we would like to generate a permutation that satisfies two properties. Based on the above intuition, (1) we want to choose a permutation that is *guaranteed* to reduce the degree of S_{rem} in each round. This way, the algorithm will end after exactly k rounds, generating exactly k permutations, where k is the degree of the initial support matrix S. (2) Second, among the permutations that will reduce the degree of S_{rem} by 1, we want to choose a permutation that will reduce the amount of remaining traffic in D_{rem} as much as possible.

Fortunately, this is a well-known optimization problem called *MWM under node coverage constraints* [19], [20], where efficient polynomial-time algorithms already exist to satisfy our two desired properties. In this formulation, we identify *critical* rows and columns in S_{rem}, which are the rows and columns that have k nonzero entries, where k is

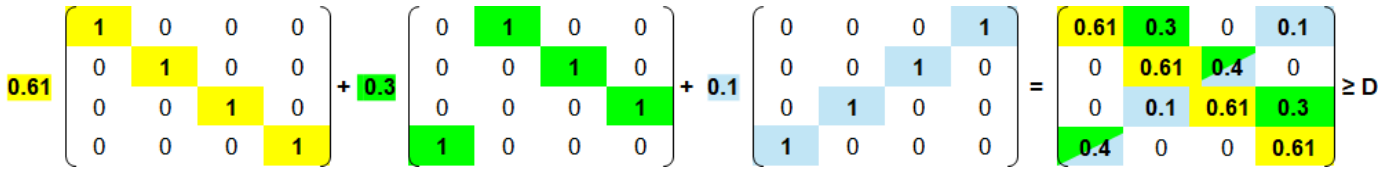


Fig. 3. Example decomposition into $k = 3$ weighted permutations that cover D.

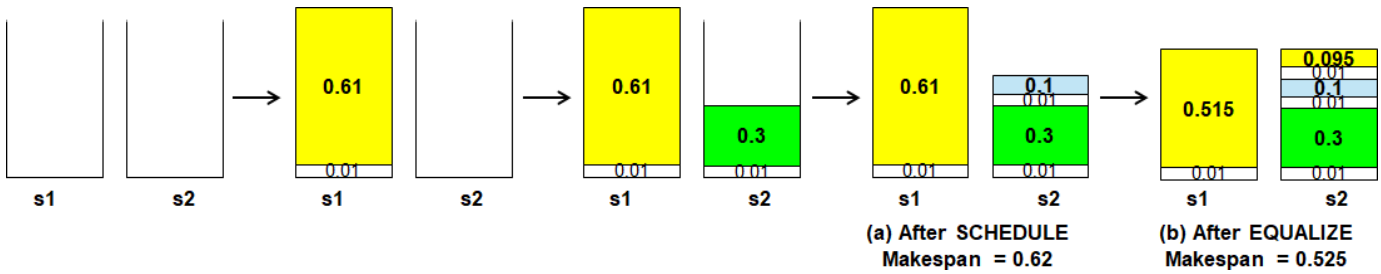


Fig. 4. (a) Scheduling the $k = 3$ permutations across $s = 2$ switches. (b) Equalizing the loads among the switches.

Algorithm 2 Subroutine REFINE

1: **Input:** D, $\mathcal{A}$ and $\mathcal{P}$
2: $\hat{\mathcal{A}} \leftarrow \mathcal{A}$
3: $D_{rem} \leftarrow D - \left(\sum_{i=1}^{k} \alpha_i P_i\right)$
4: **for** $i = 1$ to k **do**
5: $\quad d \leftarrow \max_{(a,b),(P_i)_{ab}=1} (D_{rem})_{ab}$
6: $\quad \hat{\alpha}_i \leftarrow \hat{\alpha}_i + d$
7: $\quad$ **for** all (a, b) such that $(P_i)_{ab} = 1$ **do**
8: $\quad\quad (D_{rem})_{ab} \leftarrow \max(0, (D_{rem})_{ab} - d)$
9: **return** $\hat{\mathcal{A}}$

the degree of S_{rem}. To ensure that the degree of S_{rem} gets reduced by 1, we *constrain the matching of every critical row or column to a nonzero entry in that row or column*. Among the permutations that satisfy this constrained matching, we use D_{rem} as weights to optimize the MWM. This constrained MWM problem can be solved in polynomial time using the Hungarian algorithm [19], modified to handle the above coverage constraints. This is shown in Line 4 of Alg. 1.

For the generated P_i, we derive the initial weight α_i by taking the minimum entry $(D_{rem})_{ab}$ where the corresponding entry $(P_i)_{ab}$ is a 1 (Line 5). We then update D_{rem} and S_{rem} at the end of each round accordingly (Lines 8-9), and we repeat the while-loop until all 1-elements in S are covered by a permutation in $\mathcal{P}$. Note that S_{rem} at each round corresponds to the remaining *uncovered* elements of S till that round, and not necessarily to the support of D_{rem}.

REFINE subroutine. After the while-loop in Lines 3-10 of Alg. 1, we are guaranteed that the set of permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ will cover S, but the weighted sum with the corresponding $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ may not necessarily cover D. This is because in Line 5, we are taking the *minimum* value in D_{rem} covered by P_i, but our stopping criterion is just that the initial support matrix S is covered by the set of permutations. The role of the final REFINE step in Line 11 of Alg. 1 is to refine the set of weights to ensure that the weighted sum indeed covers D.

Algorithm 3 SCHEDULE

1: **Input:** $\mathcal{A}$, $\mathcal{P}$, s and δ
2: Sort $\mathcal{P}$ by non-increasing weights in $\mathcal{A}$ such that $\alpha_{\sigma(1)} \geq \alpha_{\sigma(2)} \geq \cdots \geq \alpha_{\sigma(k)}$
3: **for** $h = 1$ to s **do**
4: $\quad L_h \leftarrow 0$, $\mathcal{P}^h \leftarrow \{\}$, $\mathcal{A}^h \leftarrow \{\}$
5: **for** $i = 1$ to k **do**
6: $\quad h^* = \text{argmin}_h\, L_h$
7: $\quad \mathcal{P}^{h^*} \leftarrow \mathcal{P}^{h^*} \cup P_{\sigma(i)}$
8: $\quad \mathcal{A}^{h^*} \leftarrow \mathcal{A}^{h^*} \cup \alpha_{\sigma(i)}$
9: $\quad L_{h^*} \leftarrow L_{h^*} + \delta + \alpha_{\sigma(i)}$
10: $\mathcal{S} \leftarrow \{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$
11: **return** $\mathcal{S}$

Given a set of k permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ that covers S, we could readily derive a set of new weights $\hat{\mathcal{A}} = \{\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_k\}$ such that the weighted sum covers D, by solving the following linear programming (LP) problem:

$$\min \sum_{i=1}^{k} \hat{\alpha}_i, \qquad \text{s.t.} \sum_{i=1}^{k} \hat{\alpha}_i P_i \geq D. \tag{5}$$

Instead, starting from the initial weights $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ from Lines 3-10, we greedily increase the weights to a new set of weights $\hat{\mathcal{A}}$ such that the weighted sum covers D (see Alg. 2). We find that in practice this greedy approach works just as well as solving the LP.

C. SCHEDULE

After the DECOMPOSE step, we have a set of permutations $\mathcal{P}$ and their corresponding weights $\hat{\mathcal{A}}$. In the SCHEDULE step, we want to assign each of these permutations to one of the s switches to minimize the makespan (Eq. (4)). This is equivalent to the makespan minimization problem on identical parallel machines, for which the *Longest Processing Time (LPT) First* algorithm [21] is a well-known and effective greedy heuristic. We adopt it for our SCHEDULE step (Alg. 3).

Algorithm 4 EQUALIZE

1: **Input:** $\mathcal{S} = \{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$, s and δ
2: **for** $h = 1$ to s **do**
3: $\quad L_h \leftarrow \sum_{i=1}^{k^h} (\delta + \alpha_i^h)$
4: cont $\leftarrow$ TRUE
5: num_iterations $\leftarrow 0$
6: **while** cont **and** num_iterations $< ks$ **do**
7: $\quad$ num_iterations $\leftarrow$ num_iterations $+ 1$
8: $\quad h_{max} = \text{argmax}_h L_h$
9: $\quad h_{min} = \text{argmin}_h L_h$
10: $\quad$ **if** $(L_{h_{max}} - L_{h_{min}}) > \delta$ **then**
11: $\quad\quad \mu \leftarrow (L_{h_{max}} + L_{h_{min}} + \delta)/2$
12: $\quad\quad z \leftarrow \text{argmax}_i \alpha_i^{h_{max}}$
13: $\quad\quad$ **if** $\alpha_z^{h_{max}} > (L_{h_{max}} - \mu)$ **then**
14: $\quad\quad\quad \tau \leftarrow L_{h_{max}} - \mu$
15: $\quad\quad\quad \alpha_z^{h_{max}} \leftarrow \alpha_z^{h_{max}} - \tau$
16: $\quad\quad\quad$ Copy permutation $P_z^{h_{max}}$ to $\mathcal{P}^{h_{min}}$
17: $\quad\quad\quad$ Add τ as corresponding weight to $\mathcal{A}^{h_{min}}$
18: $\quad\quad\quad L_{h_{max}} \leftarrow L_{h_{max}} - \tau$
19: $\quad\quad\quad L_{h_{min}} \leftarrow L_{h_{min}} + \delta + \tau$
20: $\quad\quad$ **else**
21: $\quad\quad\quad$ cont $\leftarrow$ FALSE
22: $\quad$ **else**
23: $\quad\quad$ cont $\leftarrow$ FALSE
24: $\hat{\mathcal{S}} \leftarrow$ updated $\{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$
25: **return** $\hat{\mathcal{S}}$

SCHEDULE pseudocode. In this greedy algorithm, we first sort the permutations by non-increasing weights, as shown in Line 2. We then initialize the load L_h for all switches to 0 with no permutations assigned to switch h yet, as shown in Lines 3-4. Then in Lines 5-9, following a non-increasing order of weights, we assign in a greedy manner the corresponding permutation to the least loaded switch (Line 6), add the permutation to the set of permutations already assigned to that switch (Lines 7-8), and update the load of that switch, including the reconfiguration delay δ (Line 9). In the end, the procedure returns the schedule $\mathcal{S}$ of the s switches in the form of weights and permutations $(\mathcal{A}^h, \mathcal{P}^h)$ per switch.

D. EQUALIZE

Finally, Fig. 4(b) illustrates how we can reduce the makespan by redistributing some of the switching load from one switch to another.

EQUALIZE pseudocode. The pseudocode for this EQUALIZE step is shown in Alg. 4. We first compute the loads L_h for each switch after the SCHEDULE step in Lines 2-3. We then iteratively equalize the loads in Lines 6-23 by moving a portion of the load from the most loaded switch h_{max} to the least loaded switch h_{min}. Lines 8-9 identify h_{max} and h_{min}. If the difference between them is more than δ, we compute the target load μ to be the average of the two loads that includes a δ delay when a portion of h_{max} is moved to h_{min} (Lines 10-11). In the most loaded switch h_{max}, we identify the index z of the longest-duration permutation in h_{max} in Line 12. If the weight of this permutation $\alpha_z^{h_{max}}$ is more than the difference between the current and target load of h_{max}, then we move the difference τ from $\alpha_z^{h_{max}}$ in h_{max} to h_{min} by copying the corresponding permutation $P_z^{h_{max}}$ to switch h_{min} and assigning a weight of τ to it (Lines 13-17). We then adjust the loads of the two switches accordingly in Lines 18-19. The iterations continue as long as load balancing is possible.

E. Time Complexity

DECOMPOSE relies on a loop with k iterations, based on the support matrix logic. At each iteration, the MWM-Node-Coverage takes $O(n^3)$ steps [22], yielding $O(kn^3)$ overall. In practice, the performance can be sped up on average cases through a Jonker-Volgenant implementation [23], [24]. The REFINE subroutine goes through k iterations, each corresponding to a particular weight α_i. Each iteration requires finding a maximum of up to n entries in D_{rem}, and up to n updates to D_{rem}, which results in $O(kn)$ complexity. Therefore, the total time complexity of DECOMPOSE is bounded by $O(kn^3)$.

SCHEDULE requires an $O(k \log k)$ sorting of $\mathcal{A}$. There are also k main iterations, with each iteration requiring an $\text{argmin}_h L_h$. If we store the group of L_h as a heap of size s, then an iteration will be $O(\log s)$. This yields a total of $O(k \log k + k \log s) = O(k \log(ks))$.

EQUALIZE has $\text{argmax}_h L_h$, $\text{argmin}_h L_h$, and $\text{argmax}_i \alpha_i^{h_{max}}$ in the loop. If we keep heaps to store these collections, then each loop iteration would take $O(\log k + \log s)$ steps. The total number of iterations of the loop is bounded by ks, thus the total time complexity is $O(ks \log(ks))$. In practice, EQUALIZE uses significantly fewer than ks iterations.

IV. LOWER BOUNDS

In this section, we provide new lower bounds for the scheduling makespan of any demand matrix D using s optical switches and a reconfiguration cost of $\delta > 0$.

Notations. We index the rows and columns of $n \times n$ matrix D using $i \in [1, 2n]$, such that the first n indices cover the n rows and the next n indices correspond to the n columns.

Given some row or column of D, we provide below several lower bounds using different approaches. Then, the maximum of all these lower bounds over all rows/columns is also a lower bound for the makespan.

Property 2. *Assume that an alternative demand matrix D_i constructed with the same row/column $i \in [1, 2n]$ as D and zero everywhere else yields L lower bounds $LB_i^{(l)}$ for $l \in [1, L]$ on its schedule makespan. Then a lower bound on the schedule makespan for D is*

$$LB = \max_{i \in [1,2n]} \max_{l \in [1,L]} LB_i^{(l)}. \tag{6}$$

Proof: To schedule D we need to schedule at least its row/column i, so $LB_i^{(l)}$ is a lower bound on the makespan for D. Moreover, the finite cartesian product of finite sets also has a finite cardinality; and for any finite set of finite numbers, the maximum element exists, is within that set, and is also finite.

Thus, the maximum of the lower bounds is a lower bound itself, and it will be one of these lower bounds $LB_i^{(l)}$. ■

Lower bound 1. We start by demonstrating a general lower bound for any number of elements per row/column.

Theorem 1 (Lower bound 1). *Assume that row/column i has k_i nonzero elements and a total weight w_i. Then the scheduling makespan has lower bound*

$$LB_i^{(1)} = \frac{w_i + \delta \cdot \max(k_i, s)}{s} \tag{7}$$

Proof: Any set of s schedules that we produce needs to schedule at least the k_i nonzero elements of row/column $i \in \{1, \ldots, 2n\}$. To determine a lower bound, we assume that this is all we need to service and neglect the other elements in the demand matrix. We need at least k_i scheduling configurations to schedule k_i elements. Their total weight also needs to be at least w_i. Thus, the sum of the s schedule lengths will be at least w_i plus the reconfiguration times, i.e., $w_i + k_i \cdot \delta$. Since we have s parallel schedules, the worst-case schedule duration is at least the average schedule duration, i.e., at least $\frac{w_i + k_i \cdot \delta}{s}$.

In addition, we need to schedule w_i over the s switches, yielding a w_i/s average delay, and each switch needs at least δ reconfiguration time. Thus another lower bound is $\frac{w_i}{s} + \delta$. Combining both, we get the result. ■

Example. Assume that in row/column i of doubly stochastic demand matrix D, there are $k_i = 16$ nonzero elements. If $s = 4$, then our lower bound is $LB_i^{(1)} = \frac{1+16\cdot\delta}{4} = 1/4 + 4 \cdot \delta$.

Lower bound 2. We now prove a second lower bound for the special case where the number k_i of nonzero elements in row or column i exactly equals s. We start with a useful lemma.

Lemma 1. *If row/column i has $k_i = s$ nonzero elements, and there is a total of $s+m$ configurations in the s parallel schedules, then the s parallel schedules will necessarily service at most m element chunks. In other words, at least $s - m$ whole elements will be serviced within a single configuration and will not be spread between configurations.*

Proof: We will prove the lemma by induction on m. First, if there are no reconfigurations at all ($m = 0$) and the s nonzero elements are scheduled by the s configurations of the s parallel schedules, then necessarily each element is assigned to a single distinct configuration and single distinct schedule. That is, they cannot be combined within any schedule in any way. Otherwise, if one of them were cut into two chunks that are serviced by different configurations and/or schedules, then we would have $s + 1$ chunks serviced within the s schedules. Thus, by the pigeonhole principle, at least one schedule would service at least two chunks, which would need a reconfiguration. Contradiction.

Assume that the statement holds for $m \geq 0$, and let's prove it for $m + 1 \geq 1$ reconfigurations. (1) Assume that the last reconfiguration of at least one schedule out of the s schedules is followed by (a) an element chunk or (b) a zeroed element. Then by removing the reconfiguration and (a) appending the element chunk to another chunk of this element, or (b) throwing out the zeroed element, we get a valid schedule with m reconfigurations. By the inductive hypothesis, this alternative valid schedule would have at most m chunks, and therefore we can conclude that our real schedule has at most $m + 1$ chunks. (2) Else, if only full nonzero elements follow reconfigurations, either (a) all nonzero elements are full and there are no element chunks, which proves the result, or (b) at least one schedule starts with a nonzero element chunk. Permute this nonzero element chunk with any nonzero full element that follows a reconfiguration, and we get back to case (1). Thus all cases hold for $m + 1$. ■

Theorem 2 (Lower bound 2). *Assume that row/column i has $k_i = s$ nonzero elements $x_1 \geq x_2 \geq \cdots \geq x_s$, and a total weight w_i. Then the scheduling makespan has lower bound*

$$LB_i^{(2)} = \delta + \min\left(x_1, \max\left(x_2, \frac{w_i + 1 \cdot \delta}{s}, x_s + \delta\right), \min_{2 \leq m \leq s^2} \max\left(x_{m+1}, \frac{w_i + m \cdot \delta}{s}\right)\right). \tag{8}$$

Proof: To establish the lower bound, we assume again that we only want to schedule the nonzero elements of row/column $i \in \{1, \ldots, 2n\}$, and neglect all the other elements in D. We can use Lemma 1. With no reconfigurations, all elements are serviced as whole elements (within a single configuration) and do not need to be split into several chunks. Thus the makespan is at least the weight of the strongest element x_1, since it cannot be split, together with its configuration cost δ. Likewise, with m reconfigurations, at least $s - m$ elements will be serviced within a single configuration, and only m elements can be split. In particular, at least one element of weight at least x_{m+1} will be serviced within a single configuration, so the makespan is at least $x_{m+1} + \delta$.

Assume that our optimal schedule uses m reconfigurations. The theorem successively considers all possible numbers m of reconfigurations, and the makespan beyond the initial configuration cost of δ:

(0) Assume we have no reconfiguration at all. Then we saw that the makespan is at least $\delta + x_1$. Note that adding the average weight w_i/s to δ would also be a lower bound for the makespan (as also shown in Theorem 1 using $k_i = s$), but since x_1 is the maximum weight, $x_1 \geq w_i/s$ so it forms a tighter lower bound.

(1) Assume we have exactly $m = 1$ reconfiguration. Then after the initial configuration cost of δ, the makespan is at least (a) x_2, as we saw; (b) the average schedule length, given that the total schedule length is the sum of the weight w_i and the reconfiguration delay δ, and that there are s schedules; and (c) the weight x_s of the smallest element together with the reconfiguration delay δ. This is because we saw that at most one element is cut. Consider the unique schedule with the reconfiguration delay. If both of its configurations are two chunks of the same cut element, then we can just get a better schedule by merging them and removing the reconfiguration, so this cannot happen. So at least one of the two elements in the schedule with the reconfiguration delay is not cut, thus its weight is at least x_s. To conclude with $m = 1$, since the makespan is at least all of these three options, it is also at least their maximum.

(m) Assume we have $m \geq 2$ reconfigurations. Then after the initial δ, the makespan is at least (a) x_{m+1}, as we saw; and

(b) the average schedule length, given that the total schedule length is the sum of the weight w_i and the m reconfiguration delays of δ each, and that there are s schedules. Thus it is at least the maximum of these two options.

Finally, since two chunks of the same element within the same schedule can always be merged, the optimal number of reconfigurations is no more than s^2, and therefore bounded. Moreover, an optimal number of reconfigurations with a minimal makespan could be any m, so the minimum of these values forms a lower bound. ■

Note that when its condition $k_i = s$ holds, the lower bound from Theorem 2 is at least as good as the first one from Theorem 1: $LB_i^{(2)} \geq LB_i^{(1)}$. Furthermore, it is strictly better when not all nonzero elements are equal. Specifically, the first lower bound becomes $LB_i^{(1)} = \frac{w_i}{s} + \delta$. The proof details the case with $m = 0$ reconfigurations, and in the other cases the inequality is strict as $\delta + \frac{w_i + m \cdot \delta}{s} > \delta + \frac{w_i}{s}$ for $m > 0$.

Demand matrix as a sum of weighted permutations. We are interested in a special case where the $n \times n$ demand matrix D is a sum of (weighted) permutations. This case appears for example in two workloads: (1) the benchmark workload used in the evaluations; and (2) a Fully Sharded Data Parallel (FSDP2) [25], [26] workload with HSDP configuration using random server placement: FSDP2 traffic is implemented as a hierarchical ring, and inter-server communications can be seen as the sum of several parallel rings.

In such a case, we prove in the Appendix that for a large enough n, i.e., for a large D, it is highly likely that *D's degree is k*. As a result, there is at least one row or column with exactly k nonzero elements, and therefore we can apply the lower bounds of §IV to some row/column i using the parameter k. This applies to the benchmark workload in the evaluations.

V. Evaluations

A. Settings

We now want to evaluate our SPECTRA algorithm. These evaluations have two goals. First, we evaluate SPECTRA on traffic matrices from *AI training workloads*. To do so, we use both a relatively sparse and strongly skewed matrix from a *GPT* model, as well as introduce a relatively dense and uniform matrix from a model with *Mixture of Experts (MoE)*. A second goal is to compare SPECTRA with previous state-of-the-art algorithms that were designed for similar objectives, and understand the impact of different characteristics of D. To do so, we use a standard *benchmark workload* that has already been used in the evaluations of several papers.

AI training workload 1: GPT-3B. Our first AI training workload results from a GPT model [27] with 3B parameters. Li et al. [28] use Microsoft's DeepSpeed framework [29] to apply a hybrid parallelism that combines pipeline parallelism (PP), tensor parallelism (TP) and data parallelism (DP). They also adopt the default mapping approach of DeepSpeed in a platform of 32 RTX3090 GPUs. They obtain a 32×32 total traffic matrix D that is strongly skewed and quite sparse. We then normalize D to make it doubly substochastic, and add a small Gaussian noise to nonzero elements of standard deviation $\sigma = 0.3\%$ as in the literature benchmark. Using random noise, we generate 50 random instances of D then average our results.

AI training workload 2: MoE. We generate MoE routing traces by running three training iterations of the open-source Qwen2 MoE model (57B) on Megatron-LM across 64 NVIDIA H100 GPUs, organized as 8 HGX nodes (with 80 GB per GPU). We start from its released weights, use synthetic input tokens, and record the token-to-expert routing decisions. We use 64 experts, with each expert being placed on one of the 64 GPUs. During each training iteration and each expert layer, we track communication traces between experts and accumulate each layer's traffic into a 64×64 demand matrix D, where each entry is the token count from source to destination. With 28 expert layers per iteration, we get $3 \cdot 28 = 84$ distinct traffic demand matrices D, over which we average our results. We also later approximate this MoE workload using a Zipf distribution to study its sensitivity.

Workload 3: Benchmark. We use the standard benchmark in the field [7], [8], [11]. It builds a 100×100 demand D that consists of $m = 16$ random flows per source port, including 4 large flows, which evenly split 70% of the bandwidth, and 12 small flows, which evenly split 30%, to reflect the sparse and skewed datacenter traffic. Each flow corresponds to a permutation matrix, and the sum of flows yields D. Finally, the nonzero demands are perturbed with small Gaussian noise of standard deviation 0.3% of the link bandwidth. Using random noise, we generate 50 random instances of D then average our results.

Algorithms. We want to evaluate our SPECTRA algorithm against the best state-of-the-art optical switch scheduling algorithms. Unfortunately, most algorithms in the literature consider a single optical switch and cannot be applied to scheduling across s parallel optical switches. We compare SPECTRA against several algorithms:

(1) BASELINE. The closest to our work is LESS [11], which considers scheduling a demand matrix D across s parallel optical switches by splitting D into s sub-workload matrices $D_1, \ldots, D_s$, each D_i to be independently scheduled on switch i. The splitting objective is to maximize the *sparsity* in these sub-workload matrices by minimizing the sum of nonzero elements across $D_1, \ldots, D_s$. While LESS uses a partial-reconfiguration scheduling algorithm intended for partially-reconfigurable switches, we adopt its sparsity-based approach as our BASELINE for comparisons. For an apples-to-apples comparison, we also use our DECOMPOSE algorithm to schedule the sub-workload matrices in BASELINE, since LESS does not provide a similar decomposition algorithm.

We further introduce three algorithms that only replace the DECOMPOSE step in SPECTRA while keeping the other steps, thus obtaining three SPECTRA variants:

(2) SPECTRA *(*ECLIPSE*).* This variant relies on ECLIPSE [7], a widely used algorithm in decompositions with reconfiguration delays, to replace the DECOMPOSE step in SPECTRA.

(3) SPECTRA *(*BIRKHOFF+*).* This variant relies on BIRKHOFF+ [10], a novel algorithm in contexts where

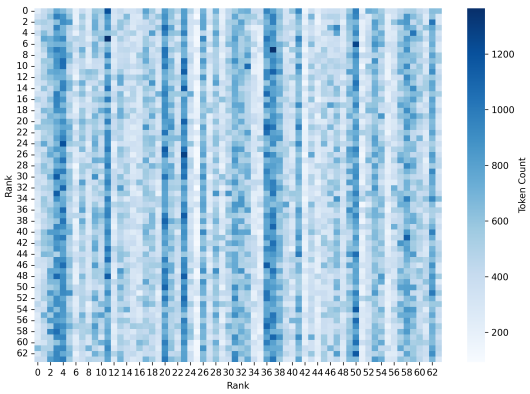


Fig. 5. MoE traffic matrix heatmap from 64 sender GPUs (rows) to 64 receiver GPUs (columns) for the first layer in the first iteration of training.

reconfiguration delay is not considered. The algorithm makes an approximate decomposition that is at most ϵ off from the true demand D's Frobenius norm. Since we want to schedule all traffic, ϵ is set to be near 0 (about 10^{-12}). Additionally, BIRKHOFF+ is designed for doubly stochastic matrices; however, our workload mostly consists of doubly substochastic matrices. In such cases, we also use von Neumann's algorithm [30], [31] that adds demand to D to make it doubly stochastic, before applying BIRKHOFF+.

(4) SPECTRA *(*WWFA*).* Chronos [32] has recently suggested using WWFA (Wrapped Wave Front Arbiter) as a greedy MWM approximation. We also examine a SPECTRA version that uses WWFA for the DECOMPOSE step.

Implementation and Runs. For the Hungarian algorithm, we implemented an efficient variant called the Jonker and Volgenant algorithm that runs faster in practice [23], [24], [33]. We conduct our experiments on a computer with a 3.7 GHz AMD Ryzen Threadripper 3970X Processor and 128 GB of memory, running on Ubuntu 20.04.6 LTS. The runtimes for SPECTRA are from under 1 ms to tens of ms.[1] Each datapoint is the average of 50 runs with random matrices, except for MoE, where it is the average over the 84 measured matrices.

B. AI Workloads

Fig. 5 shows the distribution of traffic during the first iteration and first expert layer of the Qwen model. Each GPU expert sends out 32768 tokens (row sums of the demand matrix). Each token consists of 2 bytes of data (BF16 encoding). Its traffic is relatively dense. It is slightly non-uniform between different destination experts, as we can see some heavier columns, but more dense and uniform than the GPT pattern, and slightly more uniform than some other smaller examples in the literature [2], [32], [34], [35].

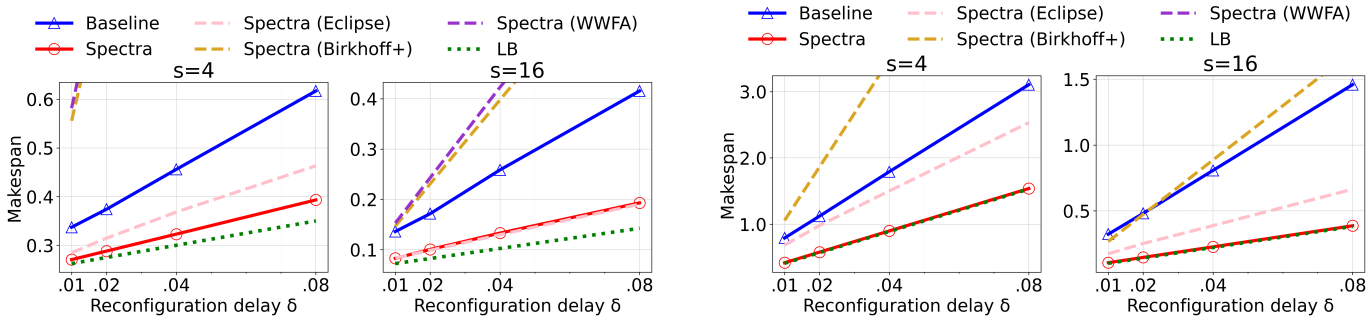
Fig. 6 presents the results for the sparse skewed GPT workload (Fig. 6(a)) and for the dense near-uniform MoE workload (Fig. 6(b)). It also shows our derived lower bound (§IV). First, SPECTRA *appears close to optimal*, since it is close to the lower bound (LB), especially with the MoE workload where it is indistinguishable, maybe because it can focus on the few columns of highest-weight destination experts (Fig. 5). Moreover, in all cases, SPECTRA far outperforms BASELINE. In fact, SPECTRA's schedule is $1.4\times$ shorter than BASELINE on average on GPT traffic, and $2.7\times$ shorter on average on MoE. Specifically, not only does SPECTRA perform much better for low reconfiguration cost δ, but also its makespan grows slower as δ grows. This is because BASELINE tries to optimize for a minimal number of nonzero elements to each switch's sub-workload demand matrix, but this optimization does not directly minimize the number of reconfigurations and does not necessarily translate into a lower makespan. Thus, this sparsity objective does not appear to fit our settings.

Using ECLIPSE for the DECOMPOSE step achieves close results with the GPT workload due to its well-structured and nearly doubly stochastic matrix, but performs much worse with the denser and substochastic MoE matrix. That is, the ECLIPSE-based variant of SPECTRA performs $1.1\times$ worse on average for GPT, and $1.7\times$ worse on average for MoE. Using BIRKHOFF+ for the DECOMPOSE step is actually worse than BASELINE for the GPT and MoE workloads, performing worse than BASELINE by $2.1\times$ for the GPT workload, and worse than BASELINE by $1.4\times$ on average for the MoE workload. It likely does not perform well because it does not take into account reconfiguration delay, and given a low ϵ, BIRKHOFF+ will decompose D into many permutations with very low weight compared to the reconfiguration delay. Using WWFA for the DECOMPOSE step attains the worst performance. In Fig. 6(b), the makespan for WWFA is so large that it had to be cropped out of the graph. WWFA is worse than BASELINE by $2.3\times$ on average for the GPT workload, and by $14.5\times$ on average for the MoE workload. This is likely because, in addition to not considering reconfiguration delays, WWFA is also greedy and does not efficiently decompose demand D into a small number of permutations. The rest of this section will thus exclude WWFA and BIRKHOFF+ from the graphs for clarity of the rest of the data. In summary, the results show that SPECTRA performs well on both sparse and dense matrices, which highlights its flexibility.

Fig. 7 analyzes the sensitivity of SPECTRA to its equalization component. We can see that when we need to schedule the dense and near-uniform MoE traffic with many small elements (Fig. 7(b)), it is relatively easy to spread the traffic, and therefore equalization does not help. However, as we use the GPT traffic with large elements (Fig. 7(a)), the elements need to be split, and *equalization makes a significant difference*.

Fig. 8 presents the impact of added noise on the performance of SPECTRA and SPECTRA with ECLIPSE in the AI training workloads. It plots the result of having a small noise added to all non-zero elements (of standard deviation $\sigma = 0.3\%$, plotted in gray) vs. having a larger noise ($\sigma = 1\%$, plotted with usual colors). The 0.3% noise is already added in the initial GPT workload, while we add it specifically in the MoE workload for the comparison. The GPT initial traffic matrix is more regular and close to being doubly stochastic, and therefore we can see that noise has a slightly negative impact, especially for $s = 4$. On the contrary, MoE traffic is strongly substochastic, so the noise can in fact make it less

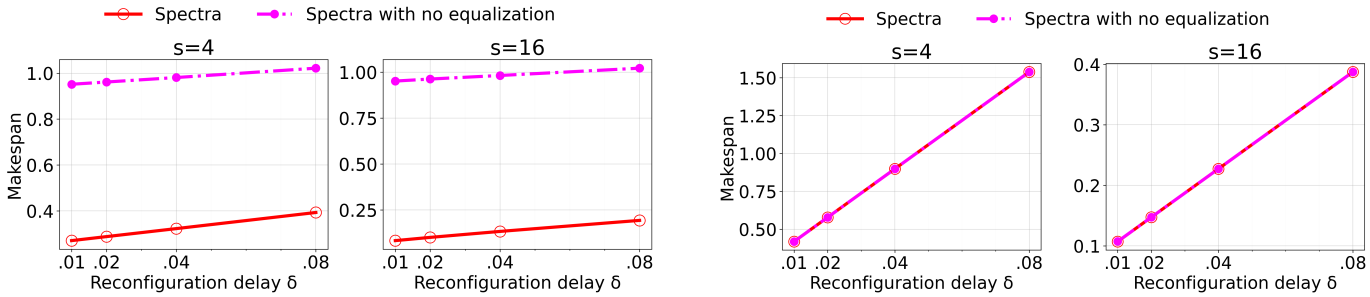
[1]Datacenter switches typically use specialized network processors or ASICs, so SPECTRA runtimes can be much faster in practice.



(a) GPT with $n = 32$.

(b) MoE with $n = 64$.

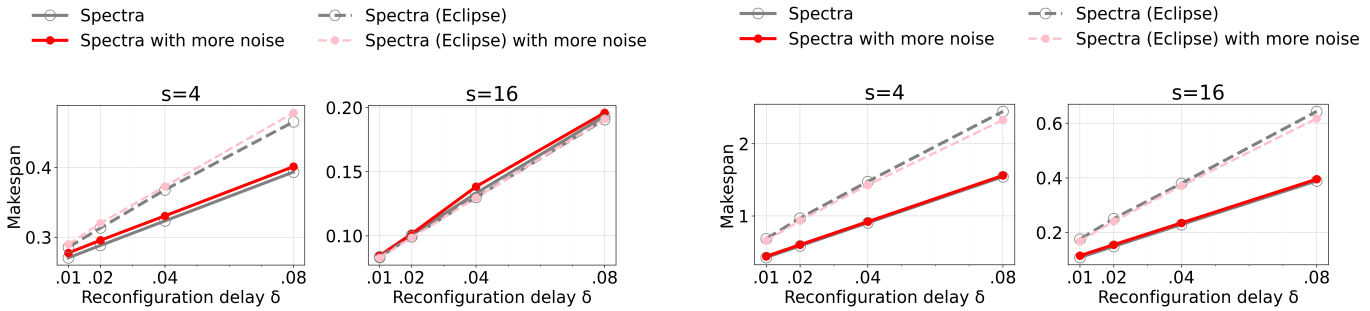
Fig. 6. Performance with *AI training workloads* given varying reconfiguration delay δ, with (a) GPT and (b) MoE workloads, using two different numbers s of OCSes.



(a) GPT with $n = 32$.

(b) MoE with $n = 64$.

Fig. 7. *Sensitivity to equalization* with *AI training workloads*, comparing SPECTRA with and without equalization.



(a) GPT with $n = 32$.

(b) MoE with $n = 64$.

Fig. 8. *Sensitivity to noise* with *AI training workload*: Comparison of SPECTRA variants with noise standard deviations 0.3% and 1%.

skewed and improve the performance.

Fig. 9 analyzes the sensitivity of the algorithms to the evolution of traffic during the 84 stages of training the MoE model. In addition to having a consistently lower makespan than the other approaches, SPECTRA also has the lowest variance and the most predictability versus the other approaches.

Fig. 10 analyzes the sensitivity of the algorithms to scaling the number n of racks, the reconfiguration delay δ, and the number s of switches. Since we want to scale to larger n, we approximate the MoE traffic matrices (which use $n = 64$) by using a synthetic traffic with a Zipf distribution of parameter 1. We also vary this parameter. We can see that as we increase the number of racks or the reconfiguration delay, SPECTRA and BASELINE have a proportional near-linear increase in makespan, with SPECTRA achieving a much better perfor-

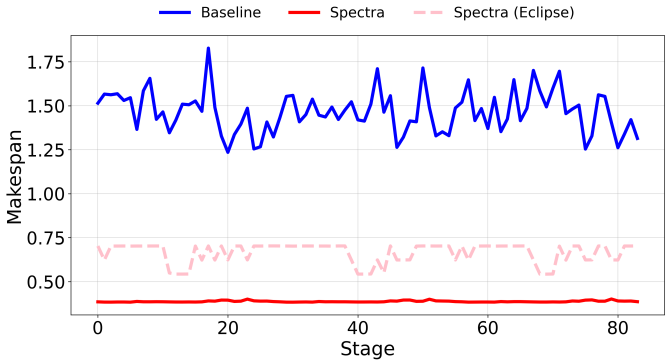


Fig. 9. *Sensitivity to traffic change* with AI training workload (MoE) across 84 stages of training.

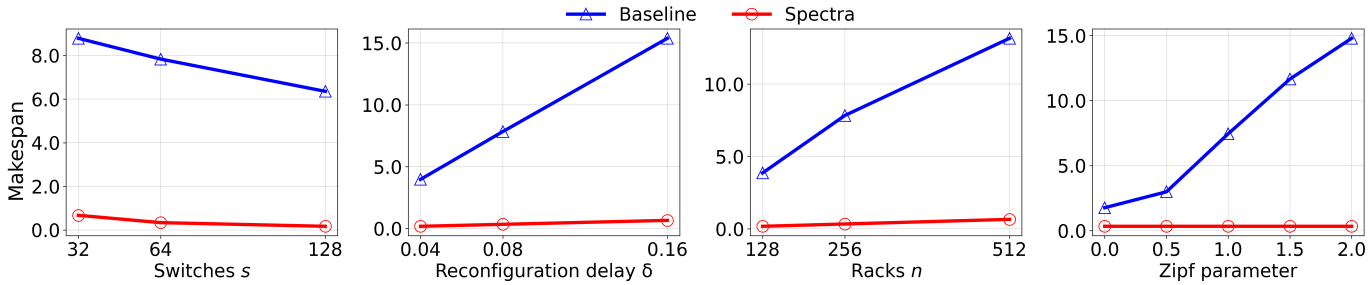


Fig. 10. *Sensitivity to scaling* with Zipf workload.

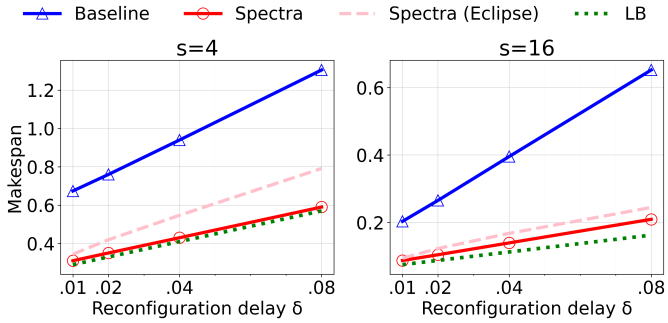


Fig. 11. *Benchmark workload.*

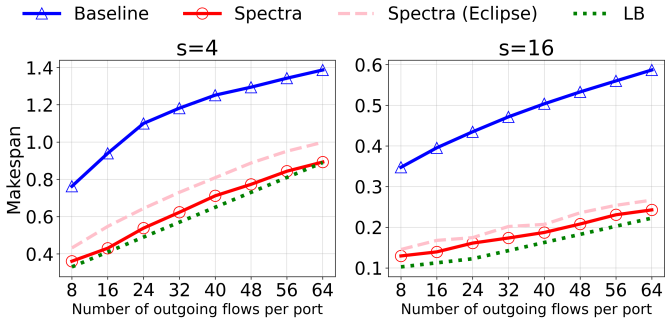


Fig. 12. *Sensitivity to sparsity* with *benchmark workload*, with varying number of flows.

mance. Likewise, SPECTRA's makespan clearly outperforms that of BASELINE as the number s of switches or the Zipf parameter vary.

C. Benchmark Workload

Fig. 11 compares SPECTRA against alternatives on the benchmark workload. SPECTRA vastly outperforms BASELINE, with a $2.4\times$ shorter schedule on average. Using ECLIPSE for the DECOMPOSE step always yields worse results, and SPECTRA performs better than SPECTRA (ECLIPSE) with a $1.2\times$ shorter schedule on average. SPECTRA also appears quite close to the lower bound, and therefore, to optimality.

Fig. 12 analyzes the sensitivity of the algorithms to a varying number m of outgoing flows per port, and therefore to a varying sparsity of the matrix D, given reconfiguration delay $\delta = 0.04$. SPECTRA is flexible enough to adapt to these differing sparsity patterns, again vastly outperforming BASELINE. Using an ECLIPSE decomposition step never helps SPECTRA.

VI. CONCLUSION

In this paper, we showed how the increasing network demands in AI datacenters could be serviced by a set of parallel OCSes, in which we want to minimize the makespan. We then explained how SPECTRA can schedule a demand matrix D over s parallel switches using three steps: DECOMPOSE, SCHEDULE and EQUALIZE. Our evaluation of SPECTRA shows that it vastly outperforms BASELINE by an average factor of $1.4\times$ on GPT AI workloads, $2.7\times$ on MoE AI workloads, and $2.4\times$ on standard benchmarks.

ACKNOWLEDGMENT

The authors would like to thank Ori Cohen for providing the MoE measurements, and Mark Silberstein and Jose Yallouz for their assistance in coordinating the necessary resources.

APPENDIX

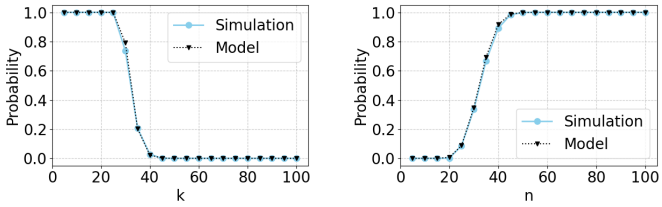
Sum of k random weighted permutations. We focus on the case where the $n \times n$ demand matrix D is the sum of k random weighted permutations, as in the benchmark workload in the evaluations (before noise is added). We want to show that for a large enough n, it is highly likely that *D's degree is k*, meaning there is at least one row or column with exactly k nonzero elements, and therefore we can apply the lower bounds of §IV to some row/column i using the parameter k. We start by proving a property of a single row or column of D.

Proposition 1. *If D is defined as the sum of k random weighted permutations, then the probability p that there are exactly k nonzero elements in a given row or column of D is*

$$p = \frac{n!}{(n-k)! \cdot n^k} \tag{9}$$

Proof: Consider k colored balls that each randomly choose one of n bins. p is the probability that they end up in distinct bins. It is the division of the number $n \cdot (n-1) \cdot \dots \cdot (n-k+1) = \frac{n!}{(n-k)!}$ of possible choices for the placement of the balls such that they get in distinct bins, by the total number n^k of possible placements. ■

We now build a model that approximates the probability that D has degree k. This model considers all rows and columns of D as i.i.d. for this approximation.



(a) k varies, given $n = 100$ (b) n varies, given $k = 16$

Fig. 13. Probability that an $n \times n$ matrix D that is the sum of k random weighted permutations also has degree k.

Proposition 2. *If D is defined as the sum of k random weighted permutations, then the probability that its degree is k (and therefore the lower bounds for i hold with k nonzero elements) can be approximated as:*

$$1 - \left(1 - \frac{n!}{(n-k)! \cdot n^k}\right)^{2n}. \tag{10}$$

Proof: We approximate the $2n$ rows and columns as i.i.d., with a probability p that any given row or column has k nonzero elements. Then the probability that at least one row or column has k nonzero elements is $1 - (1-p)^{2n}$. ■

Fig. 13(a) illustrates the simulated probability that the degree of D is k in a 100×100 matrix D that is the sum of k random weighted permutations. It shows that the approximation is close. Fig. 13(b) shows the same probability for a sum of 16 permutations, as D's size n varies. Again, the approximation holds well. Moreover, for n beyond 50, it is highly likely that its degree will be 16.

References

[1] K. Liang, L. Qiao, I. Keslassy, and B. Lin, "Scheduling parallel optical switches in datacenters," *IEEE Networking Letters*, vol. 8, pp. 149–153, 2026.

[2] X. Liao, Y. Sun, H. Tian, X. Wan, Y. Jin, Z. Wang, Z. Ren, X. Huang, W. Li, K. F. Tse *et al.*, "MixNet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training," in *ACM SIGCOMM*, 2025, pp. 554–574.

[3] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang *et al.*, "RDMA over Ethernet for distributed training at Meta scale," in *ACM SIGCOMM*, 2024, pp. 57–70.

[4] J. Lu, J. Gao, F. Feng, Z. He, M. Zheng, K. Liu, J. He, B. Liao, S. Xu, K. Sun *et al.*, "Alibaba Stellar: A new generation RDMA network for cloud AI," in *ACM SIGCOMM*, 2025, pp. 453–466.

[5] K. Qian, Y. Xi, J. Cao, J. Gao, Y. Xu, Y. Guan, B. Fu, X. Shi, F. Zhu, R. Miao *et al.*, "Alibaba HPN: A data center network for large language model training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 691–706.

[6] Y. Feng, T. Chen, Y. Wei, S. Shen, S. Wang, W. Li, K. Ma, and T. Hoefler, "RailX: a flexible, scalable, and low-cost network architecture for hyper-scale LLM training systems," *arXiv preprint arXiv:2507.18889*, 2025.

[7] S. Bojja Venkatakrishnan, M. Alizadeh, and P. Viswanath, "Costly circuits, submodular schedules and approximate carathéodory theorems," *Queueing Systems*, vol. 88, no. 3, pp. 311–347, 2018.

[8] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *ACM CoNEXT*, 2015, pp. 1–13.

[9] N. Farrington *et al.*, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *ACM SIGCOMM*, vol. 40, no. 4, 2010, pp. 339–350.

[10] V. Valls, G. Iosifidis, and L. Tassiulas, "Birkhoff's decomposition revisited: Sparse scheduling for high-speed circuit switches," *IEEE/ACM Transactions on Networking*, vol. 29, no. 6, pp. 2399–2412, 2021.

[11] L. Liu, J. J. Xu, and M. Singh, "LESS: A matrix split and balance algorithm for parallel circuit (optical) or hybrid data center switching and more," in *IEEE/ACM Utility and Cloud Computing (UCC)*, 2019, pp. 187–197.

[12] C. Liang *et al.*, "NegotiaToR: towards a simple yet effective on-demand reconfigurable datacenter network," in *ACM SIGCOMM*, 2024.

[13] Y. Zu, A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, A. Kolbasov, A. Rezaei, D. Du *et al.*, "Resiliency at scale: Managing Google's TPUv4 machine learning supercomputer," in *Usenix NSDI*, 2024, pp. 761–774.

[14] F. De Marchi, J. Li, Y. Zhang, W. Bai, and Y. Xia, "Unlocking superior performance in reconfigurable data center networks with credit-based transport," in *ACM SIGCOMM*, 2025, pp. 842–860.

[15] V. Addanki *et al.*, "Vermilion: A traffic-aware reconfigurable optical interconnect with formal throughput guarantees," in *Usenix NSDI*, 2027.

[16] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen *et al.*, "Sirius: A flat datacenter network with nanosecond optical switching," in *ACM SIGCOMM*, 2020, pp. 782–797.

[17] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch, "TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *Usenix NSDI*, 2023, pp. 739–767.

[18] X. Li and M. Hamdi, "On scheduling optical packet switches with reconfiguration delay," *IEEE JSAC*, vol. 21, no. 7, pp. 1156–1164, 2003.

[19] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*, ser. Algorithms and Combinatorics. Berlin: Springer, 2003, vol. 24, no. 2.

[20] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[21] P. Bergé, M. Chaikovskaia, J.-P. Gayon, and A. Quilliot, "Approximation algorithm for job scheduling with reconfigurable resources," in *International Symposium on Combinatorial Optimization*, 2024, pp. 263–276.

[22] H. N. Gabow, "Data structures for weighted matching and extensions to b-matching and f-factors," *ACM Transactions on Algorithms (TALG)*, vol. 14, no. 3, pp. 1–80, 2018.

[23] R. Jonker and A. Volgenant, "Linear assignment procedures," *European journal of operational research*, vol. 116, no. 1, pp. 233–234, 1999.

[24] ——, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, no. 4, pp. 325–340, 1987.

[25] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[26] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "PyTorch FSDP: Experiences on scaling fully sharded data parallel," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3848–3860, 2023.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[28] W. Li, X. Liu, Y. Li, Y. Jin, H. Tian, Z. Zhong, G. Liu, Y. Zhang, and K. Chen, "Understanding communication characteristics of distributed training," in *Proceedings of the 8th Asia-Pacific Workshop on Networking*, 2024, pp. 1–8.

[29] "Megatron-DeepSpeed," https://github.com/microsoft/Megatron-DeepSpeed.

[30] J. von Neumann, "A certain zero-sum two-person game equivalent to the optimal assignment problem," in *Contributions to the Theory of Games II*, ser. Annals of Mathematics Studies. Princeton University Press, 1953, vol. 28, pp. 5–12.

[31] C.-S. Chang, W.-J. Chen, and H.-Y. Huang, "Birkhoff-von Neumann input buffered crossbar switches," in *IEEE Infocom*, 2000, pp. 1614–1623.

[32] S. Renganathan and N. McKeown, "Chronos: Prescheduled circuit switching for LLM training," in *Proceedings of the 2nd Workshop on Networks for AI Computing*, 2025, pp. 89–97.

[33] D. F. Crouse, "On implementing 2D rectangular assignment algorithms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 4, pp. 1679–1696, 2016.

[34] S. Gandhi and C. Kozyrakis, "MoEtion: Efficient and reliable checkpointing for mixture-of-experts models at scale," *arXiv preprint arXiv:2412.15411*, 2024.

[35] E. Cohen, J. Yallouz, A. Shpiner, M. Silberstein, S. Ratnasamy, and I. Keslassy, "Incast-free MoE rate-based scheduling," *arXiv preprint arXiv:2607.26340*, 2026.