EFFICIENT OPTIMIZATION TECHNIQUES FOR TRAFFIC ENGINEERING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Firas Abuzaid
August 2022

This dissertation is online at: https://purl.stanford.edu/nj221cf1818

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Peter Bailis, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Balaji Prabhakar

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Many enterprises today manage their Internet traffic on their wide-area networks (WANs) using centralized, software-defined traffic engineering (TE) schemes. These schemes, however, scale poorly with network size; as the network grows, the runtimes required to determine flows on the network scale super-linearly. In response, network operators often fall back on simple heuristics to meet their SLAs. Unfortunately, these heuristics come at the expense of optimality: they lead to inexact, approximate solutions that can lead to poor utilization on the network. The status quo is thus a difficult trade-off: fast solutions that are significantly sub-optimal, or optimal solutions that are too slow.

However, these WAN topologies can be *partitioned*, either geographically or by their resources (i.e., their commodities and link capacities). We analyze real-world traffic from the Microsoft Azure WAN and demonstrate that its traffic is often clustered; this insight motivates us to leverage partitioning to design novel, more scalable algorithms for traffic engineering. Finding a valid partitioning is non-trivial, because we must adhere to the constraints of the original traffic engineering problem, such as demand constraints and capacity constraints. At the same time, the partitioning cannot be designed poorly; otherwise it could yield substantial approximation errors in the resultant flow allocations. We show that, by cleverly designing our partitioning strategies, we can develop algorithms that provide a better trade-off between runtime and optimality.

First, we propose NCFlow, which builds off of the geographic partitioning strategy. Instead of solving a global flow problem on the entire WAN, NCFlow solves *i)* a simpler problem on a contraction of the network, and *ii)* a set of sub-problems in parallel on disjoint clusters within the network. Our results on the topology and demands from the Microsoft Azure WAN, as well as on publicly available topologies, show that NCFlow nearly matches

the solution quality of currently deployed solutions for the maximum total flow objective (99.1% of optimality in the median case), but is $11\times$ faster than the state of the art, with maximum observed speedups of $1,900\times$. Moreover, NCFlow outperforms other heuristics and approximation methods and realizes a better trade-off between optimality and runtime.

Second, we propose POP, an alternative algorithm that leverages commodity-based partitioning. This approach exploits the Law of Large Numbers and *randomly* splits the TE problem into smaller, independent sub-problems: each sub-problem executes the original flow problem on the same topology, but with a fraction of the WAN's link capacities, as well as a subset of the commodities. We demonstrate that, by *reusing* the original flow problem, POP can generalize to many TE objectives, not just maximum total flow. We also provide theoretical and empirical evidence to justify random partitioning as an effective strategy. In our experiments across the same set of topologies and demands, we show that, in the median case, POP realizes 99.9% of the optimal solution, but is $18\times$ faster than the state of the art, with maximum observed speedups of $98\times$.

In the third part of this dissertation, we compare and contrast NCFlow and POP and discuss their relative strengths and weaknesses. We show that NCFlow excels when the optimal approach requires a small subset of the commodities to take all the capacity, which directly contrasts with POP. Finally, we conclude by discussing how POP can be applied to problems beyond traffic engineering and extend to other large-scale computer systems problems, such as cluster scheduling and load balancing.

Acknowledgements

"Not all those who wander are lost" – J.R.R. Tolkien

It may sound cliché, but the Ph.D. is truly a journey, in every sense of the word. The work over the past seven years that ultimately culminated in this document has been long and arduous, with many ups and downs. There were periods of wandering, during which I *did* feel lost. To be candid, I often wondered during those periods whether or not I would make it to the finish line.

Now that the end is in sight, I find the quote above to be particularly appropriate. I realize that this insight—that those who wander are not necessarily lost—is probably the most valuable lesson I have learned from my experience as a Ph.D. student. And, above all else, I am deeply grateful to every person along the way who has helped me find my way and finally reach the end of this rewarding journey.

First and foremost, I owe an enormous debt of gratitude to my advisors, Peter Bailis and Matei Zaharia, as well as Samuel Madden, one of my original advisors. Without them, I wouldn't have even been able to start this journey, much less complete it. When I first began the Ph.D., I was a student at MIT, not Stanford, advised by both Matei and Sam. As it just so happened, Sam had invited a recent graduate from UC Berkeley to take a postdoctoral position in his group: Peter.

As soon as I began working with him, I understood that I had found a special mentor. Peter was, simply put, the most enthusiastic person I had ever met in real life—he was Tony Robbins with a doctorate degree in Computer Science. His enthusiasm was contagious; when I worked with him, I would immediately notice how much my own motivation increased simply by being in his presence. He showed me that passion is actually a skill,

that it can be honed and cultivated, just like any other talent, and I have tried to take that lesson with me in all of my other endeavors in life. Peter taught me many valuable lessons about the art of research, but the most critical lesson he passed on to me was much more fundamental: the importance of taking care of myself. When I started my Ph.D., I began to struggle with a lot of daily stress and anxiety. Peter noticed this and made a suggestion that changed my life: he recommended that I see a mental health therapist. With his encouragement (and despite my trepidation), I made an appointment with a counselor the very next week, and I haven't looked back ever since—it has certainly improved my life beyond measure. Thank you, Peter, for making me a better researcher and a healthier, happier person.

Matei has been an incredible research mentor to me throughout my entire journey. He has been insightful, patient, and wise every step of the way. While Peter exuded energy, Matei was always zen, and I can't thank them enough for establishing this happy medium for me. I must confess that I had a bit of a wandering eye during my Ph.D.; I struggled to pick a single topic and stick to it. Ultimately, my dithering probably slowed me down, but Matei was supportive through thick and thin. No matter what research topic I picked—whether it was systems and machine learning, data analytics, artificial intelligence in medicine, or (finally) computer networking—he was always willing to dive in. Moreover, he always had a good idea up his sleeve, a nugget of wisdom or unique perspective into the problem that he could share. His breadth of knowledge is truly remarkable, and he has served as an examplar for me in that regard. And, despite this brilliance, Matei has also been arguably the most humble person I have ever worked with. Unfortunately, brilliance and humility do not go hand in hand in our society, and we accept that trade-off as a *de facto* rule of human nature. Matei has proven to be the exception to that rule, and I am truly, truly grateful for that. I am extremely lucky that I was able to work with him over these past seven years.

I am grateful to Balaji Prabhakar, Keith Winstein, and Stephen Boyd for serving on my defense committee and improving my research in the process. Balaji was kind enough to invite me to present some of my work to his research group in 2020, and he also served on my reading committee for this dissertation. Keith provided me with excellent feedback when I first proposed my thesis, and I have long admired his incisiveness and mental clarity

when discussing networking, computer systems, or, frankly, any other topic. (If you ever get the chance, just ask him about emojis. I promise that you won't be disappointed.) Stephen chaired my defense, but he was also a co-author on one of the papers featured in this dissertation. His zeal for research and depth of expertise has always been inspirational to me, to say the least.

In 2019, during one of those moments when I was wandering in the wilderness, I decided to apply for an internship at Microsoft Research. On the receiving end of that application was Srikanth Kandula, who, for some reason, was willing to take a massive risk on a student who had no research experience in computer networking whatsoever. To this day, I still do not know why he agreed to take me on; all I know is that it turned out to be one of the best things that ever happened to me during my Ph.D. Srikanth helped me rediscover my passion for research—he lit a fire under me that summer that I desperately needed. We spent hours together pair programming at his desk and even longer at the whiteboard, dissecting our proposed algorithms well into the night to anticipate what could go wrong. I had a front-row seat to observe all the traits that constitute an excellent researcher: a breadth of knowledge that spanned theory to systems and everything in between; a meticulousness that both left me in awe and drove me insane; and an indefatigable work ethic that never waned, no matter how challenging the situation was. Most of all, he forced me—in a good way—to expand my abilities as a researcher; he challenged me to really *own* my research. Prior to that summer internship, I was often plagued by self-doubt in my research abilities. Srikanth provided me with the confidence I needed to evolve as a researcher and finally take on that sense of ownership I was lacking. After that wonderful summer, I continued to work with him remotely from Stanford, and we even reconnected in person for a three-week working session in early 2020, right before the pandemic began. Needless to say, he features prominently in the work presented in this dissertation. Thank you, Srikanth, for being such a brilliant mentor to me. I only wish we had begun working together sooner.

While at Microsoft, I was also fortunate enough to work with Behnaz Arzani and Ishai Menache, both of whom were excellent collaborators and co-authors. Behnaz was exceptionally sharp in all of our brainstorming sessions, and I always appreciated her ability and willingness to dive deep with me into the nitty-gritty of the mathematical optimization literature, which helped me get up to speed on a new research topic that I had no experience in.

Her unique expertise enriched our work and took it the next level. Ishai was the seasoned sage of our collaboration, and he always helped me understand how our work fit into the broader picture in computer networking. He was always able to find connections to other related work and articulate the subtle nuances to me, which also accelerated my onboarding into this new field. Most importantly, both Behnaz and Ishai were excellent colleagues to me that summer, amiable and approachable from day one. Thank you both so much for making me feel at home in Building 99; your hospitality will not be forgotten.

I would like to thank my many collaborators and co-authors, without whom the work in this dissertation would certainly have not been realized. A heartfelt thank you to Akshay Agrawal, Albert Rogers, Ameet Talwalkar, Andrew Feng, Anojan Selvalingam, Asvin Ananthanarayan, Atul Shenoy, Behnaz Arzani, Ce Zhang, Cesare Corrado, Christian Meyer, Christopher Ré, Daniel Kang, David Krummen, Deepak Narayanan, Edward Gan, Eric Xu, Erik Meijer, Feynman Liang, Fiodar Kazhamiaka, Geet Sethi, Ishai Menache, Jeff Naughton, Jialin Ding, John Emmons, John Sheu, Joseph Bradley, Junaid Zaman, Kexin Rong, Lee Yang, Mahmood Alhusseini, Matei Zaharia, Paul Clopton, Paul Wang, Peter Bailis, Peter Kraft, Sahaana Suri, Samuel Madden, Sanjiv Narayan, Shoumik Palkar, Srikanth Kandula, Stefan Hadjis, Stephen Boyd, Steven Niederer, Tina Baykaner, Wayne Giles, Wouter-Jan Rappel, and Xi Wu.

Thank you to my friends outside of work for providing me with a spark of joy whenever I needed it the most and keeping me sane in the process: Abdallah AbuHashem, Abdul-Kareem Agunbiade, Abi Raja, Adil Kalam, Ahmad Ibrahim, Al-Karim Lalani, Alborz Bejnood, Amine Mhedhbi, Anum Afzal, Ayesha Rasheed, Cyrus Pinto, Dahlia Fateen, Eyuel Tessema, Farah Ereiqat, Fatima Wagdy, Galym Imanbayev, Hosniya Zarabi, Ibrahim Elshamy, Jared Quincy Davis, Joe Maguire, John Emmons, Kamil Saeid, Khalil Ramadi, Mahmood AlHusseini, Malak Abu Sharkh, Mehmet Seflek, Michael Gummelt, Mohammad Usama "Juni" Khalil, Mohamed Farid, Mohammed Elasmai, Naser Dehaibi, Natalie Jabbar, Navid Chowdhury, Nishant Jacob, Omair Khan, Omar Shakir, Osama El-Gabalawy, Paroma Varma, Pukar Hamal, Rafid Sikder, Salahodeen Abdul-Kafi, Saleh Abbas, Samer Bu Jawdeh, Sanna Ali, Sarah Rangwala, Sophia Nguyen, Sughra Ahmed, Umayah Abdennabi, Yasmin Chebbi, Yifei Huang, Yuliya Mykhaylovska, Zaid Adhami, and many countless others.

When I moved from MIT to Stanford, I did not realize that I was joining a new research group called FutureData, led by both Peter and Matei. At the time, it was just a handful of second-year students that occupied a single office. But, even from its infancy, the group has always been welcoming and warm to everyone, which I have always deeply appreciated. The lab's unique culture is a primary reason why, today, it showcases a breadth of expertise—from machine learning to systems, theory to practice—that is quite rare in academia. I have been fortunate enough to work with Cody Coleman, Daniel Kang, Deepti Raghavan, Edward Gan, Fiodar Kazhamiaka, Gina Yuan, Jialin Ding, Kai Sheng Tai, Keshav Santhanam, Kexin Rong (my MSR officemate for the summer!), Omar Khattab, Peter Kraft, Sahaana Suri, and Trevor Gale in various capacities over the last seven years. Thank you all for the wisdom and knowledge you shared with me, the trenchant feedback you gave me on my research, and the delightful ski trips and overseas adventures we had together at various conferences. Even though the Ph.D. is a solo journey, it was always comforting to know that you were also charting your own path next to me, side by side.

It is difficult to put into words how much my officemates in Gates 432 have meant to me over these last seven years. Deepak Narayanan, Shoumik Palkar, and James Thomas were more than colleagues to me—they were true friends, the best friends a struggling Ph.D. student could ever ask for. When I reflect back on our time together, my mind always goes back to a famous line from *The Office*: "I wish there was a way to know you're in 'the good old days' before you've actually left them." Alas, those "good old days" are behind us, but I do have a treasure trove of invaluable memories from our time in 432 that always brings a smile to my face. I will never forget the hilarious hijinks we conducted in that office: Shoumik accidentally crashing his toy drone into the corner; James pacing around the room, twirling his pen, rapping under his breath whatever hip-hop lyrics were en vogue at the time; Deepak finishing up his hard day's worth of labor, closing his terminal, then peeling his fifth banana of the day and tuning in to watch the Boston Red Sox game. Most of all, I'll never forget the laughter—how often we laughed, how loud we laughed, how much we annoyed our neighbors with our incessant laughing. (Sorry about that!) Thank you so much for always brightening my day over these last seven years, even during the toughest times that plagued my Ph.D. I will always cherish those moments we had together in Gates 432.

And, of course, I would be remiss if I didn't give a shout-out to Pratiksha Thaker, our honorary member of 432, who was and still is always welcome to come work in our office.

Last, but certainly not least: the biggest thank you a human being could ever offer to my family. To my aunts, uncles, cousins, nieces, nephews, and grandparents: thank you for offering your love and support over the past seven years, even from thousands of miles away in Palestine.

To my mom, my dad, and my sister: thank you for absolutely everything you have done for me since I was born—for feeding me, protecting me, clothing me, nurturing me, educating me, guiding me, humbling me, and supporting me. Thank you for every sacrifice you made, for encouraging me when I was right and correcting me when I was wrong. I am so lucky to be your son, to be your sibling, and without you, I certainly would have been lost on this journey. But, with you, I've made it to the finish line with the biggest smile on my face. I couldn't have done it with you.

To Mama, Baba, and Ayyoush

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the early 1990s, the Internet has evolved from an avant-garde experiment into a mature, ubiquitous technology that has transformed our lives in every imaginable way. Today, approximately 150,000 GB of traffic is sent on the Internet every second around the world [3], and one would be hard-pressed to name another technology or innovation that has had a greater daily impact on human society.

What makes the Internet a truly global phenomenon is a concept called Wide-Area Networks (WANs), which connect Internet-accessible devices across entire towns, cities, states, and even countries. To route Internet traffic around the world, WANs have historically relied on dynamic, decentralized protocols, such as IS-IS [57, 92, 112] and BGP [10, 34, 51, 52], to coordinate and ensure that packets can be delivered safely and reliably from any source to any destination. These protocols effectively connect WANs to one another, thus making the Internet a true network of networks. They enable a user in, say, Singapore to load a website hosted on servers in Buenos Aires, for example.

However, in the last decade, Wide-Area Networks *themselves* have become global networks on their own. With the rise of cloud computing and the demand for globally available services, large-scale web companies—such as Google, Amazon, and Microsoft—have deployed datacenters all over the world, and they have constructed trans-continental WANs to ensure that these datacenters are always connected to one another.

Figure 1.1: The Microsoft Azure WAN from January 2021 [6]. For simplicity, both the "edge" WAN (which handles ISP-facing traffic) and the inter-datacenter WAN are both captured in a single map; in practice, their traffic is managed separately.

Because of their importance, these datacenter WANs have grown faster than their ISP-facing counterparts; to support their high usage, enterprises provision their links with high-capacity optical fiber. Moreover, because these WANs must connect datacenters across continents, enterprises must also lease intercontinental submarine cables to be used as links in the WAN to realize their global connectivity aspirations [81]. Unsurprisingly, datacenter WANs have become an expensive resource for cloud providers [5, 71, 72, 76]: Microsoft values their Azure WAN at a billion dollars and estimates its annual maintenance cost to be a hundred million dollars [84].

In this dissertation, we argue that routing traffic on these global datacenter WANs is a non-trivial problem for network operators: the state-of-the-art methods available today scale poorly with network size. To address this challenge, we introduce two new algorithms that can more efficiently and effectively route traffic, thereby maximizing utilization on the WAN and ensuring its costs are well worth the benefits.

1.1 Traffic Engineering on WANs

Because of their enormous value, enterprises place significant importance on traffic engineering (TE) [20]—the study of efficiently routing dynamic traffic demands—on their WANs. Network operators leverage TE to achieve high utilization. However, the traditional TE strategies, which typically prescribe decentralized routing (e.g., RSVP-TE [19,21,98]), pose a challenge for large-scale WANs. In the decentralized model, no entity has a global view of the entire set of demands on the WAN, and instead, individual sites in the network greedily select paths for their traffic using algorithms such as Constrained Shortest Path First (CSPF) [55, 56, 58, 138]. As a result, the network can get stuck in locally optimal routing patterns that are globally sub-optimal [115].

Starting about a decade ago, large cloud providers began applying the principles of software-defined networking [35,36] to TE, thus birthing a *centralized* approach for routing traffic [71, 72, 76]. Specifically, they computed optimal routing schemes for the current set of demands by solving global multi-commodity flow problems on the entire WAN [71, 72, 76]; once computed, the routes are encoded into the switch forwarding tables across the network [5, 36] (e.g., using MPLS [98]). In the centralized model, flow problems are often expressed as linear programs, with well-defined objective functions and constraints [24,29]. This gives network operators the ability to explicitly model which objectives they care about (e.g., maximizing throughput, or minimizing maximum link utilization) and define the optimal solution (and therefore measure any gaps in optimality). Most importantly, these linear programs can be efficiently solved using off-the-shelf blackbox optimization solvers, such as Gurobi [68], Mosek [7], CPLEX [45], and many others [50]. Thanks to this paradigm shift, enterprises were now able to offer low latency and high bandwidth for critical, customer-facing applications [54, 125, 143], as well as fast response times for bulk data transfers [79, 89, 117].

The centralized strategy has yielded significant improvements in utilization and overall network performance since its adoption. But, as topology sizes have continued to grow, solving multi-commodity flow problems as linear programs on the entire network has become impractical and intractable. Network operators at Google captured this point eloquently in [72]; they noted that the "algorithm run time increased super-linearly with the

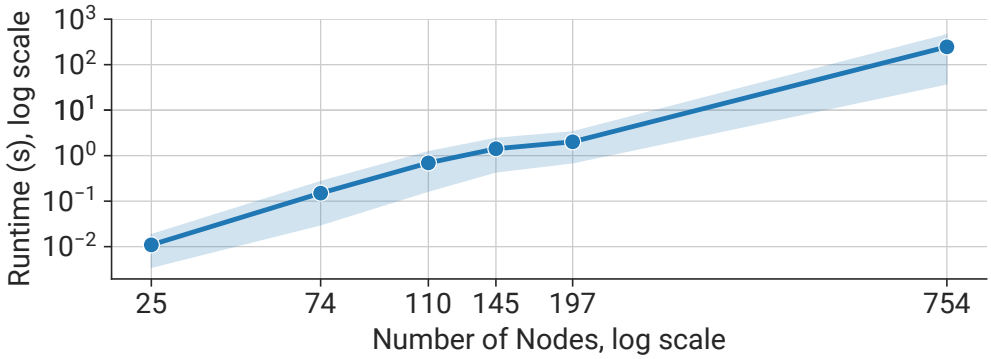


Figure 1.2: As WAN topologies grow, multi-commodity flow problems become increasingly intractable. To illustrate this, we benchmark a path-based formulation of the Multi-Commodity Maximum Flow (MCMF) linear program in Gurobi [68] on several topologies from the Internet Topology Zoo [82]. As the topology size increases, the solver runtime increases quadratically. (Note the log scale on both axes.)

site count," which led to "extended periods of traffic blackholing during data plane failures, ultimately violating our availability targets," as well as "scaling pressure on limited space in switch forwarding tables." Even when leveraging state-of-the-art optimization software, solver runtimes have increased super-linearly with topology sizes. (See Figure 1.2 for empirical evidence.)

What compounds the scalability challenge is the increasingly dynamic nature of TE in the 21st century: simply put, WAN traffic patterns have become more volatile as traffic demands have increased. For example, network operators at Facebook have observed high variability on their edge networks as recently as March 2019 [41], with daily peak traffic reaching 7× more than the trough. This dynamism is not simply a function of user behavior, however; traffic patterns may also change because of link/switch failures in the network, which can occur at a moment's notice [27, 84, 96, 147] and with surprisingly high frequency [81].

In response, enterprises have introduced tighter SLAs (e.g., within minutes) to promptly respond to traffic spikes. The ultimate upshot of these trends is this: traffic engineering has

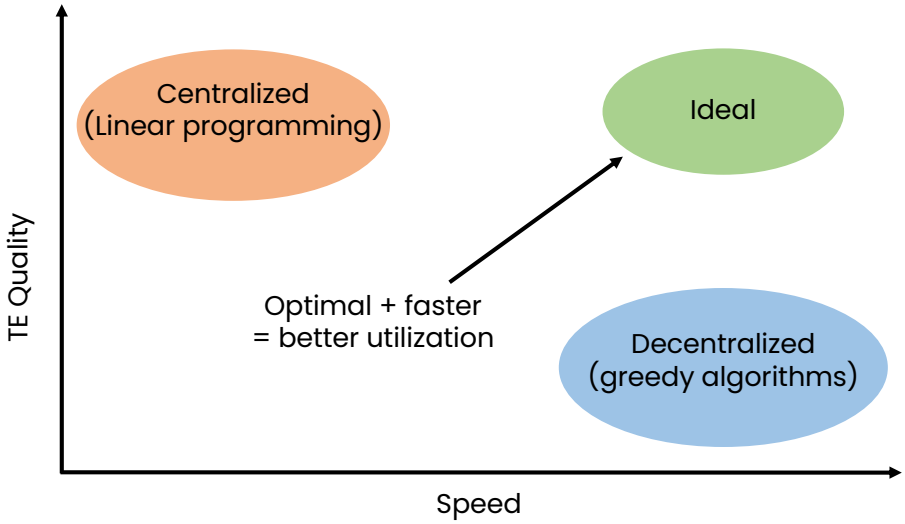


Figure 1.3: Trade-off space for traffic engineering solutions today. Linear programs produce high-quality traffic routes, but cannot keep up with today's SLAs. On the other hand, heuristic approaches and approximate algorithms are runtime-efficient, but produce sub-optimal routing. A solution that is both fast *and* optimal will lead to better utilization on the WAN.

now become a rapid-paced, high-stakes, multi-shot game, where traffic routes must be re-computed more quickly, more frequently, and more optimally than ever before. And, as illustrated in Figure 1.3, the trade-off between centralized linear programming and decentralized greedy algorithms is an unenviable predicament for network operators to stomach—they are caught between the proverbial rock and a hard place.

1.2 Partitioning the WAN

We must find a better trade-off, a solution that is both runtime-efficient and close to optimal in its routed flows. We know empirically that, for large-scale WANs with hundreds of sites, the centralized approach will not scale. On the other hand, we also know that a completely decentralized approach—where every switch is making locally optimal decisions—will lead to a collectively bad outcome as well. The question becomes: can we do something in between, a more Goldilocks solution that hits the sweet spot between complete centralization and full decentralization?

To achieve this goal, we will leverage a key idea:

We can partition WAN topologies into multiple discrete components, and these components can be treated as distinct sub-problems of the original TE problem.

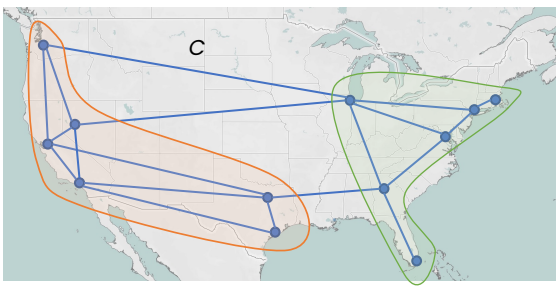
In this dissertation, we propose partitioning the WAN to transform the global TE problem into a discrete set of independent, parallelizable sub-problems. Intuitively, if we can find a clever partitioning strategy that can reliably instantiate these sub-problems, we can then use a divide-and-conquer approach to compute flow allocations and route traffic on the WAN. The added benefit from partitioning is that, if we can now decide the number of partitions, then that provides us with a tunable knob, which we can use to trade off between flow quality and runtime. Essentially, if the centralized and decentralized approaches represent two poles of the traffic engineering spectrum, partitioning allows us to navigate the entire spectrum and find that sweet spot.

Thesis statement: *By partitioning the WAN, we can use a divide-and-conquer strategy for traffic engineering to navigate the trade-off between solver runtime and traffic quality.*

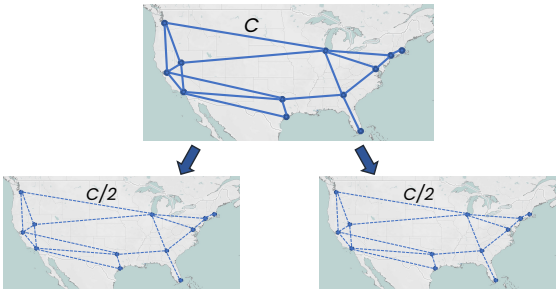
Devising a partitioning strategy for TE, however, is non-trivial. For it to succeed, it must create sub-problems that have some semblance of independence—otherwise, the problem cannot be parallelized. At the same time, the partitioning strategy must also ensure that the global problem's constraints are never violated. This transformation could (and most likely will) lead to some approximation errors for certain inputs; we will occasionally sacrifice traffic in the WAN for better runtime performance. But, if designed correctly, we will achieve a much better trade-off between optimality and efficiency.

1.3 Traffic Engineering via Partitioning: NCFlow and POP

We present two different partitioning strategies for TE: geographic partitioning and commodity-based partitioning. In geographic partitioning, we compute clusters on the network, segmenting the topology into contiguous components. Figure 1.4a shows an example of such



(a) Geographic partitioning. The C denotes the bandwidth capacity for each link in the network.



(b) Commodity-based partitioning. The link capacity C from the original topology is now $C/2$.

Figure 1.4: Examples of geographic and commodity-based partitioning of a simple WAN that spans the continental United States.

a clustering on a fictitious WAN in the continental U.S. Computing these clusters is a non-trivial matter, and a "bad" clustering can significantly impact flow quality on the network. We show that certain graph clustering algorithms that aim to preserve the densely connected sub-graphs in the topology, such as modularity-based clustering [31, 42] and spectral clustering [107, 137], are effective for traffic engineering.

To leverage geographic partitioning, we propose an algorithm called NCFlow.[1] Instead of solving the global multi-commodity flow problem on the entire WAN, NCFlow solves *i)* a simpler routing problem on the contraction of the network (i.e., the clustered graph), then *ii)* the set of routing sub-problems on disjoint clusters within the network. NCFlow still uses linear programming to solve the flow problem and maximize the total flow objective, but it modifies the linear programs to take advantage of the clustering. Through careful construction, NCFlow can solve the sub-problems in parallel, while still routing inter-cluster traffic and reconciling flows between clusters.

In our second approach—commodity-based partitioning—we *randomly* partition the TE problem into smaller, independent sub-problems: each sub-problem has the same topology, but with a fraction of the link capacities of the original problem, as well as a subset of the commodities.[2] Figure 1.4b shows an example of commodity-based partitioning on the same fictitious WAN from Figure 1.4a. We show, both theoretically and empirically, that

[1] short for **N**etwork **C**ontractions for **Flow** problems

[2] A commodity is a source-destination pair in the WAN that has requested traffic demand; we will define it more formally in Chapter 2.

random partitioning is also effective for developing a divide-and-conquer solution to TE.

To take advantage of commodity-based partitioning, we propose POP,[3] our second algorithm for scalable traffic engineering. Because of the partition is generated randomly, POP exploits the Law of Large Numbers, which yields benefits at larger scales. Our approach also allows us to *reuse* the original linear program without modification, which therefore provides more generality than the previous approach—it can be applied to more objectives beyond maximum total flow.

It is important to note that, while both NCFlow and POP use partitioning to accelerate TE, neither requires any *physical* modification to the underlying topology structure of the WAN. Both geographic and commodity-based partitioning are purely logical abstractions that we use to accelerate traffic engineering.

1.4 Summary of Results

We benchmarked both NCFlow and POP on a thorough test harness of inputs: eight publicly available WAN topologies from the Internet Topology Zoo [82], several different traffic models, and many demand scales to simulate both light and heavy traffic scenarios. Additionally, we were fortunate enough to evaluate our algorithms on a month's worth of real-world traffic demands from the Microsoft Azure WAN topology. Our results show that both NCFlow and POP achieve a better trade-off between optimality and runtime than the current state of the art in traffic engineering. In the median case, NCFlow nearly matches the solution quality of currently deployed solutions for the maximum total flow objective (99.1% of optimality), but is $11\times$ faster than the gold standard approach, with maximum observed speedups of $1,900\times$. Similarly in the median case, POP realizes 99.9% of the optimal solution, but is $18\times$ faster than the gold standard approach, with maximum observed speedups of $98\times$. In our real-world experiments, we observed similar behavior: NCFlow achieved 98.5% of optimality with $7.9\times$ speedups in the median case, while POP achieved 99.9% of optimality with $12.5\times$ speedups in the median case.

Additionally, we benchmark NCFlow and POP on real-world traffic matrices from Microsoft. we find that NCFlow is $8.5\times$ faster in the median case, while achieving 98.5%

[3] short for Partitioned Optimization Problems

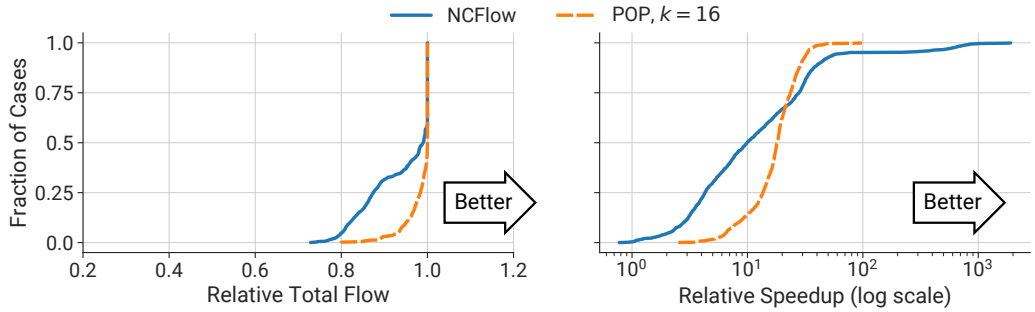


Figure 1.5: Performance of NCFlow and POP relative to a Path-Based MCMF implemented and executed in Gurobi. The left-hand side shows total volume of allocated flow; the right-hand side shows relative speedup. NCFlow is $11\times$ faster than in the median case, while POP is $18\times$ faster.

of optimality; POP is $12.5\times$ faster in the median case, while achieving 99.9% of optimality. Beyond these empirical results, we outline best-case and worst-case scenarios for both NCFlow and POP and delve into the strengths and weaknesses of each technique to characterize *how* we are able to achieve these speedups while still maximizing flow quality.

1.5 Previously Published Work

This dissertation features the following previously published work:

- **Contracting Wide-area Network Topologies to Solve Flow Problems Quickly** [12].
  Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, Peter Bailis. *NSDI 2021*.

- **Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP** [102].
  Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, Matei Zaharia. *SOSP 2021*.

1.6 Dissertation Plan

This dissertation is organized as follows: Chapter 2 introduces relevant background on traffic engineering and partitioning to appropriately ground this dissertation. Chapter 3

introduces the NCFlow algorithm; Chapter 4 introduces the POP algorithm. In Chapter 5, we compare NCFlow to POP and discuss their relative strengths and weaknesses. Chapter 6 discusses POP's applicability to broader problems in computer systems research beyond traffic engineering, including cluster scheduling and query load balancing. Finally, we conclude and discuss possible areas of future work in Chapter 7.

Chapter 2

Background on Traffic Engineering

In this chapter, we provide background and motivation on traffic engineering on Wide-Area Networks. We describe our problem setup, discuss prior solutions, and introduce notation and definitions that will be used throughout this dissertation, specifically in Chapters 3 and 4.

2.1 Historical Context

A WAN is broadly defined as any Internet network that covers a wide geographic area, and its origins can be traced to the infancy of the Internet itself: the original ARPANET first spanned the western United States before expanding across the continental U.S. Today, most, if not all, Internet Service Providers (ISPs) are considered WANs.

Over the last two decades, global Internet companies have constructed their own private datacenters around the world to better service their users. For these enterprises, classical decentralized routing protocols, such as BGP [10, 34, 51, 52], have proven to be ineffective, since they are not capacity- nor performance-aware [125]. Furthermore, research has shown that significant barriers must be overcome to improve these protocols for today's traffic demands [40, 124].

Instead, these companies have elected to maintain their own private WANs to connect their datacenters. In fact, they often build multiple WANs: some that connect to ISPs to

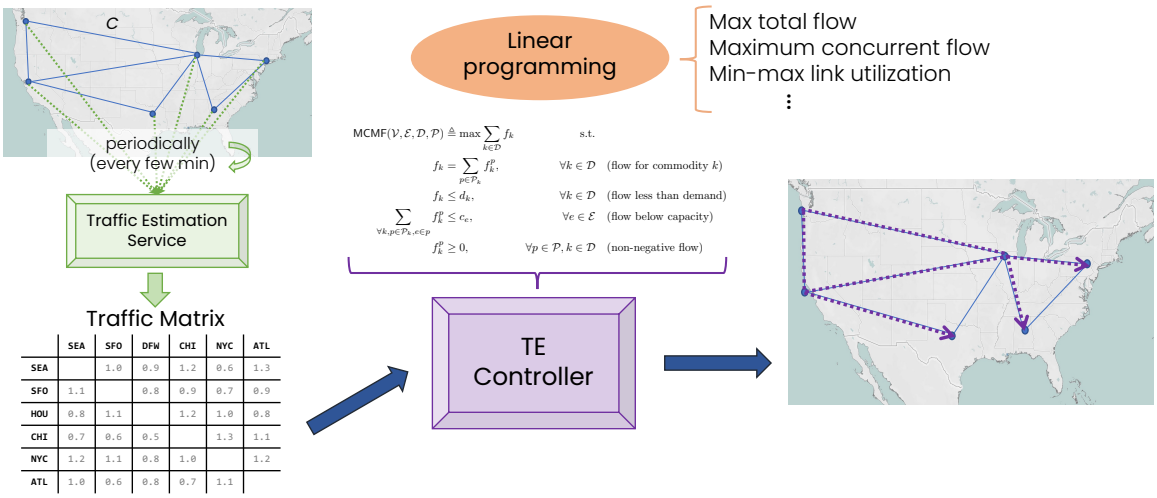


Figure 2.1: Overview of the Traffic Engineering problem for Wide-Area Networks. A traffic estimation service periodically generates a traffic matrix (TM); both the TM and the WAN topology are passed as inputs to the centralized TE controller, which is responsible for computing optimal routes for every entry in the matrix. TE controllers typically use linear programming with blackbox solvers to compute these routes; this approach provides network operators the flexibility to apply different objectives (e.g., maximum total flow, maximum concurrent flow, etc.), depending on their priorities.

reach end users [41, 125, 128, 143],[1] and others that are dedicated solely to carrying traffic *between* datacenters [72, 76]. (Figure 1.1 shows a map that displays both types of WANs operated by Microsoft for their Azure cloud service.) The reasons for this development are two-fold: *i)* inter-datacenter WANs typically exhibit unique traffic characteristics compared to user-facing traffic [71], and *ii)* traffic on these WANs makes up an increasingly large fraction of global Internet traffic today [88].

2.2 Problem Setup

In our setting, we're given as input a Wide-Area Network topology and a **traffic matrix** (TM), which represents the traffic demands on the network at a given point in time. A **commodity** is any source-destination pair in the TM (i.e., a row and a column), and a **demand** is the amount of traffic a commodity requests (i.e., the entry in the matrix for a

[1] These are often referred to as "edge" WANs in the literature.

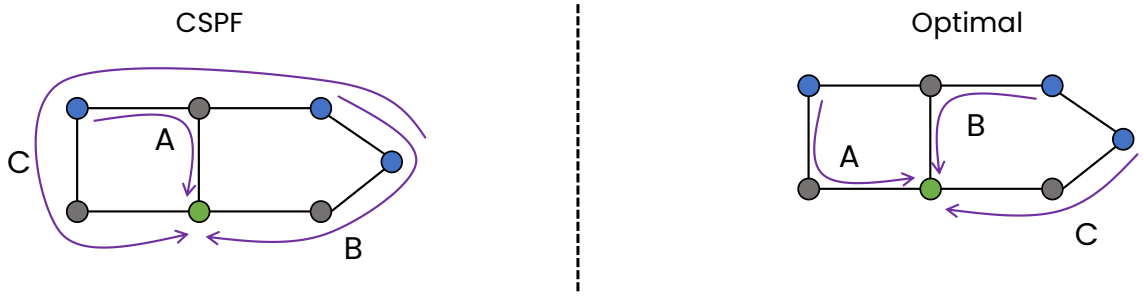


Figure 2.2: Simple example of sub-optimal routing with Constrained Shortest Path First (CSPF). Commodities A, B, and C all have unit demand and share the same destination, and each edge has unit capacity. Compared to the optimal routing on the right-hand side, CSPF's routing on the left consumes $5\times$ as many links.

row-column pair).

The traffic matrices are generated by a centralized traffic estimation service, which runs periodically on a regular interval (e.g., every five minutes). Depending on the WAN's traffic volatility, this interval window can be adjusted by the network operator, so long as the TE controller can keep up. The service passively collects sFlow [63, 116] samples exported by each network device in the WAN, then aggregates these samples per source/destination pair, applies smoothing over the time window, and produces a complete matrix for the TE controller to consume [6].

The TE controller, which is also a centralized service, has a single responsibility: compute optimal routes for every commodity's demand in the traffic matrix.

2.3 Prior Solutions

The earliest strategies for traffic engineering relied on decentralized approaches, drawing inspiration from the naturally decentralization that was characteristic of the early Internet. In this setting, no entity has a global view of the traffic demands on the WAN; instead, each individual switch decide on its own how to route traffic, by greedily selecting paths for their traffic. The most common path selection algorithm is Constrained Shortest Path First (CSPF [55, 56, 58, 138]), which essentially runs a shortest-path algorithm after pruning the links that violate the minimum bandwidth capacity constraints for the traffic. Figure 2.2

illustrates an example of how this heuristic can go awry and allocate traffic along paths that are globally sub-optimal. Prior work has shown that CSPF does not scale well to large-scale datacenter WANs, especially on the order of hundreds or thousands of sites and links [115]. Other heuristics, such as Equal-Cost Multi-Path (ECMP) routing, behave similarly [39, 73, 132].

One possible approach to cope with such massive scale is to construct the WAN topology as a hierarchy, such that only a small, well-connected core of the topology is globally managed while the rest of the WAN uses distributed heuristics [72]. Doing so adds constraints to the WAN topology, and can complicate deployment; moreover, such distributed heuristics have been shown to be sub-optimal [71].

2.3.1 Multi-Commodity Flow Problems

In this section, we give some background on multi-commodity flow problems. Given a set of nodes, capacitated edges, and demands between nodes, a flow allocation is feasible if it satisfies demand and capacity constraints. The goal of a multi-commodity flow problem is to find a feasible flow which optimizes a given objective; Table 2.2 lists some examples.

The fastest algorithms [53,80] are approximate; i.e., given a parameter ϵ, they achieve at least $(1-\epsilon)\times$ the optimal value. However, their runtime complexity is at least quadratic (see Table 2.2). Moreover, these solutions allow demands to travel on any edge, thus requiring millions of forwarding table entries at each switch for thousand-node topologies.

Instead, production systems [71, 76] restrict flow to a small number of pre-configured paths per commodity, which reduces the required forwarding table entries by 10–100$\times$.

Using notation from Table 2.1, the feasible flow over a pre-configured set of paths can be defined as:

Terms	Definitions
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, commodities, and paths
N, M, K	The numbers of nodes, edges, commodities, and paths, i.e., $N = \lvert\mathcal{V}\rvert, M = \lvert\mathcal{E}\rvert, K = \lvert\mathcal{D}\rvert$
e, c_e, p	Edge e has capacity c_e; path p is a set of connected edges
(s_k, t_k, d_k)	Each commodity k in $\mathcal{D}$ has source and target nodes ($s_k, t_k \in \mathcal{V}$) and a non-negative demand d_k.
$\mathbf{f}, f_k^p$	Flow assignment vector for a set of commodities, and the flow for commodity k on path p.

Table 2.1: Notation for framing multi-commodity flow problems.

$$\begin{aligned}
\mathsf{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq \big\{ f_k \mid \forall k \in \mathcal{D} \text{ and } & \qquad (2.1)\\
f_k = \sum_{p \in \mathcal{P}_k} f_k^p, \qquad & \forall k \in \mathcal{D} \quad \text{(flow for commodity } k\text{)}\\
f_k \le d_k, \qquad & \forall k \in \mathcal{D} \quad \text{(flow below demand)}\\
\sum_{\forall k, p \in \mathcal{P}_k, e \in p} f_k^p \le c_e, \qquad & \forall e \in \mathcal{E} \quad \text{(flow below capacity)}\\
f_k^p \ge 0 \qquad & \forall p \in \mathcal{P}, k \in \mathcal{D} \quad \text{(non-negative flow)} \quad \big\}
\end{aligned}$$

With this definition, we can define multiple objectives that we may wish to optimize for in the WAN. For example, maximizing the total flow in the network can be expressed as:

$$\begin{aligned}
\mathsf{MaxFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq \max_{\mathbf{f}} \sum_{k \in \mathcal{D}} f_k \qquad & (2.2)\\
\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) &
\end{aligned}$$

Production SDN-based TE systems at large enterprises use linear optimization-based solvers [71, 72, 76]. On WANs with thousands of nodes, the optimization problem could have millions of variables and equations just to verify that a flow allocation is feasible.[2]

[2]Specifically, over K variables and $K + M$ equations; if $N > 10^3$, $K > 10^6$

	Objective function	Additional constraints	Used in	Best known complexity
Max Total Flow	$\max \sum_{i \in \mathcal{D}} f_k$	None	[71, 76]	$O(M^2 \epsilon^{-2} \log^{O(1)} M)$ [53]
Max Concurrent Flow	$\max \alpha$	$d_k \alpha \leq f_k, \forall k \in \mathcal{D}$	[27, 77, 79]	$O(\epsilon^{-2}(M^2 + KN) \log^{O(1)} M)$ [80]
Min-Max Link Utilization	$\min z$	$\sum_{\forall k, p \in \mathcal{P}_k, e \in p} f_k^p \leq z, \forall e \in \mathcal{E}$	[86]	$O(\epsilon^{-2} \gamma K \log^{O(1)} M)$ [49]

Table 2.2: We illustrate a few different multi-commodity flow problems, all of which can be defined over FeasibleFlow in Equation 2.1 but optimize for different objectives and can have additional constraints; see notation in Table 2.1. More problems are discussed in [15].

In addition to repeatedly solving global optimizations, these TE schemes must maintain an up-to-date view of the topology, gather desired volumes for demands and update traffic splits at switches based on the result of the optimization.

Our production experience is that most of these repetitive steps have a latency of a few RTTs (round trip times); therefore, solving the optimization dominates, especially on large topologies. Moreover, demands are limited to their allocated rates in software at the source servers and thus allocating less than the full desired rate need not result in packet loss [71]. Finally, applications that contribute a large fraction of the bytes moving between datacenters are elastic in short timescales (e.g., large dataset transfers for data analytics). That is, these apps seek a fast completion time but do not need a large rate in every optimization epoch. Some other applications have a decreasing marginal utility as their rate allocation increases such as video streams of varying quality [85]. Today's SDN-based TE solutions [71, 76] use multiple priority classes to maximize allocations for elastic traffic without affecting the latency-sensitive traffic.

2.4 Changing Demands: Analysis of Production Traffic

In the previous chapter, we showed the scalability challenges of traffic engineering on publicly available WAN topologies via a simple experiment that measured TE solver runtime vs. topology size (see Figure 1.2). We also claimed that the *changes* in demands—the dynamic nature of today's traffic—can also lead to poor utilization. A natural question arises: just how much do we observe in today's datacenter WANs?

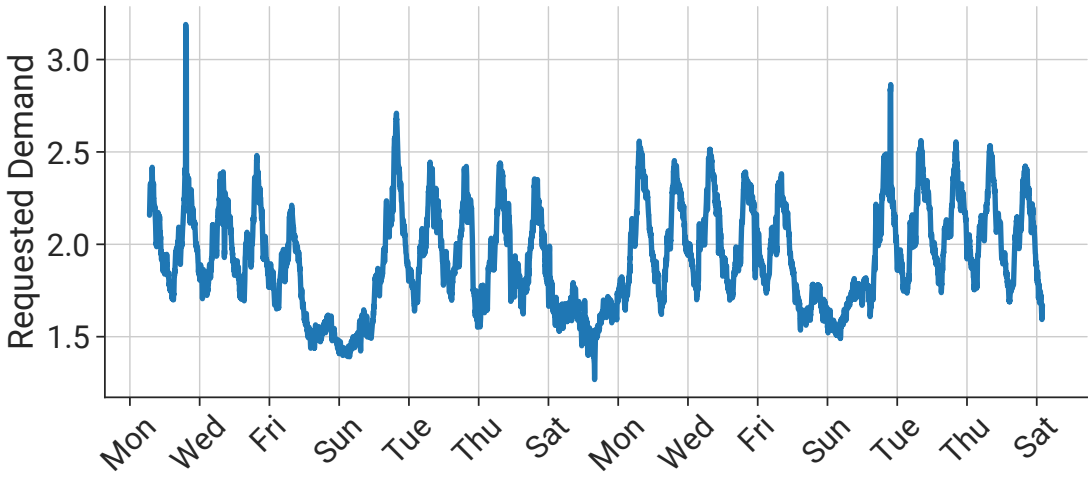


Figure 2.3: Production traffic on a Microsoft datacenter WAN over a several-month period. For privacy considerations, we plot the normalized demand, rather than the absolute traffic volume over time. The change in demand between any two successive traffic matrices is substantial: the average change is 35%, and, in 20% of the cases, the delta is over 45%.

To answer this question, we obtained and analyzed the traffic matrices from a production WAN at Microsoft over a several-month period from 2019. Figure 2.3 plots the normalized demand over time. As the Figure demonstrates, the change in traffic demand from one 5-minute window to the next is substantial: the average change is 35%, and, in 20% of the cases, the demand is over 45%. Microsoft solves a global flow allocation problem every few minutes, and we analyzed the traffic that will remain unsatisfied if the flow allocation from the previous window were to be used instead of computing a new allocation. We see that the median loss is 13%; in 20% of the cases, over 20% of the demand remains unsatisfied. We verify that computing a new allocation will satisfy all of the demand; using the previous window's allocation causes loss because some datacenter pairs may receive more flow in the previous allocation than their current demand, while other datacenter pairs go unsatisfied. Given this data, computing a new allocation in each time window is needed to carry more traffic on the WAN.

Chapter 3

NCFlow

3.1 Introduction

In this chapter, we discuss NCFlow, a more scalable algorithm for traffic engineering that leverages geographic partitioning. NCFlow is specifically designed for the maximum total flow objective; we show that, compared to the state of the art, NCFlow is substantially faster at the expense of a small amount of flow. By using a faster solver like NCFlow, WAN operators can reduce loss when faults occur and carry more traffic on the network by tracking demand changes.

Our solution is motivated by the observation that WAN topologies and demands are *concentrated*: the topology typically has well-connected portions separated by a few, lower-capacity edges, and more demand is between nearby datacenters. This is likely due to multiple operational considerations: *i)* submarine cables have become shared choke points for connectivity between continents [60, 81](see Figure 3.1), *ii)* the connectivity over land follows the road or rail networks along which fiber is typically laid out, and *iii)* enterprises build datacenters close to users, then steer traffic to nearby datacenters [16, 125, 143]. Therefore, more capacity and demand are available between *nearby* nodes; an analysis of data from a production WAN at Microsoft in §3.2 supports this observation.

We leverage this concentration of capacity and demand to partition the WAN geographically, thereby decomposing the global flow problem into several smaller problems, many of which can be solved in parallel. As shown in Figure 3.3, we divide the network into

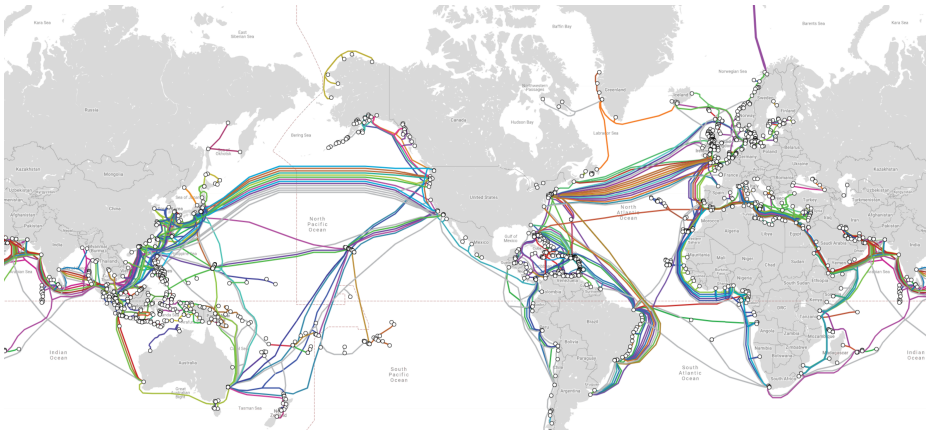


Figure 3.1: Submarine cables serve as choke points in global datacenter WAN topologies; figure is excerpted from [126].

multiple connected components, which we refer to as *clusters*. We then solve modified flow problems on each cluster, as well as on the *contracted network*, where nodes are clusters and edges connect clusters that have connected nodes. Prior work [1, 11, 23] notes that Google and other map providers use different contractions to compute shortest paths on road network graphs. Our goal is to closely match the multi-commodity max flow solution in quality (i.e., carry nearly as much total flow), while reducing the solver runtime and number of required forwarding entries. We discuss related work in §3.7; to our knowledge, we are the first to demonstrate a practical technique for multi-commodity flow problems on large WAN topologies.

Solving flow problems on the contracted network poses two key challenges:

1. How to partition the network into clusters? More clusters leads to greater parallelism, but maximizing the inter-cluster flow requires careful coordination between the sub-problems at multiple clusters.
2. How to design the sub-problems for each cluster to improve speed while reducing inconsistencies in allocation? The sub-problem for a cluster has fewer nodes and edges to consider, but it will not be be faster if it must consider all node pairs whose traffic can pass through the cluster.

Our solution NCFlow achieves a high-quality flow allocation with a low runtime and

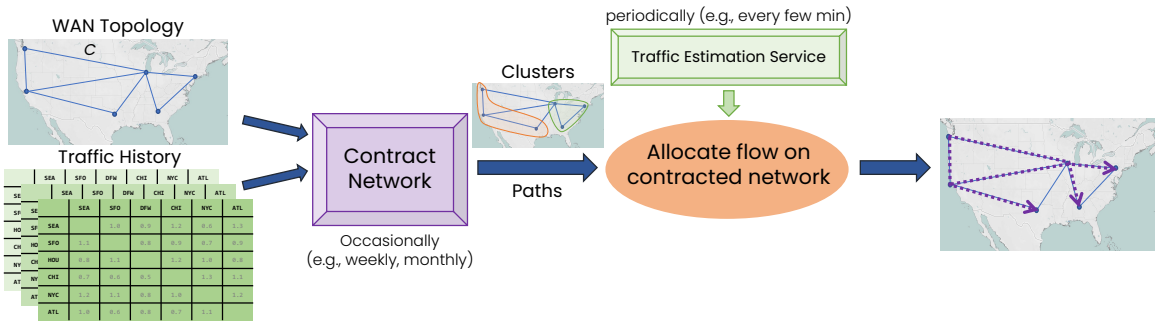


Figure 3.2: NCFlow's workflow. Our proposed solution first computes clusters on the WAN topology, based on the topology itself and the history of traffic demands. (The latter is optional and depends on the clustering algorithm's required inputs.) In addition to the clusters, a set of cluster-aware paths are also computed. Then, for each new traffic matrix, NCFlow allocates flows based on the clusters and paths from the previous step.

space complexity by addressing each of these challenges in turn. First, we contract the network using well-studied algorithms such as modularity-based clustering [42] and spectral clustering [107], which are designed to identify the choke-point edges in a network. Second, we *bundle* commodities whose sources and/or targets are in the same cluster, treating them as a single commodity. In Figure 3.3 for example, when routing traffic from source nodes in the red cluster to target nodes in the green cluster, the yellow cluster treats that traffic as a single bundled commodity, instead of (up to) 24 individual commodities. Doing so can lead to inconsistent flow allocations between clusters (which we explain in §3.3.1), and we devise careful heuristics to provably avoid them (§3.3.2). Finally, we show that bundling demands between clusters provides an additional benefit to WAN operators: we can reduce the number of forwarding entries needed at switches by reusing pathlets within clusters and traffic splitting rules across multiple demands (§3.3.5).

Figure 3.2 shows the workflow for NCFlow. First, we choose appropriate clusters and paths using an offline procedure over historical traffic—these choices are pushed into the switch forwarding entries. This step happens infrequently, such as when the topology and/or traffic changes substantially. Then, online (e.g., once every few minutes), NCFlow computes how best to route the traffic over the clusters and paths, similar to deployed solutions [71, 72, 76].

Overall, our key contributions are:

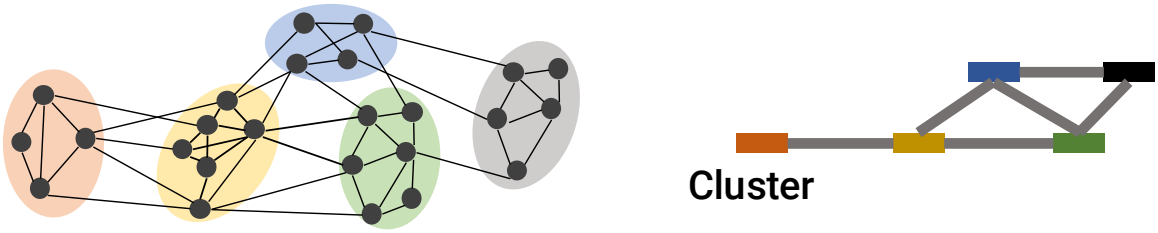


Figure 3.3: Example clustering of a WAN to produce a contracted network for NCFlow. The original network on the left is divided into clusters, shown with different background colors. The contracted network is on the right.

- We propose NCFlow, a decomposition of the multi-commodity max flow problem into an offline clustering step and an online, provably feasible, algorithm that solves a set of smaller sub-problems in parallel.
- We evaluate NCFlow using real traffic on a large enterprise WAN, as well as synthetic traffic on eleven topologies from the Internet Topology Zoo [82]. Our results show that, for multi-commodity max flow, NCFlow is within 2% of the total flow allocated by state-of-the-art path-based LP solvers [71, 72, 76] in 50% of cases; NCFlow is within 20% in 97% of cases. Furthermore, NCFlow is at least $8\times$ faster than path-based LP solvers in the median case; in 20% of cases, NCFlow is over $30\times$ faster. Lastly, NCFlow requires 2.7–16.7$\times$ fewer forwarding entries in the evaluated topologies. NCFlow also compares favorably to state-of-the-art approximation algorithms [53, 80] and oblivious techniques [86, 118].
- We show that, as a fast approximate solver, NCFlow can be used to react quickly to demand changes and link failures. Specifically, in comparison to TEAVAR [27], NCFlow carries more flow when no faults occur and suffers about the same amount of total loss during failures.

We have open-sourced NCFlow at https://github.com/stanford-futuredata/pop-ncflow.

3.2 Background and Motivation

In this section, we provide summary of the main findings which have influenced NCFlow and shaped our design choices. Our observation that demand and capacity are concentrated among *nearby* nodes in the WAN topology is grounded on the following measurements from a production WAN at Microsoft:

Demand properties:

- On average, 7% (or 16%) of the node pairs account for half (or 75%) of the total demand.
- When nodes are divided into a few tens of clusters, 47% of the total traffic stays within clusters. If the demands were distributed uniformly across node pairs, only 8% of the traffic would stay within clusters; thus the demand within clusters is about $6\times$ larger than would be expected from a uniform distribution.

WAN topology properties:

- When nodes are divided into tens of clusters, 76% of all edges and 87% of total capacity is within clusters.
- The skew in capacity is small: the ratio between the largest edge capacity and the mean is 10.4.
- The skew in node degree is also small: the average node degree is 3.9, with $\sigma = 2.6$; the max is 16.
- Relative to the network size (hundreds of nodes), the average network diameter (=11) and the average shortest-path length ($= 5.3$) are very small.

Motivated by the above analyses, NCFlow seeks to be a fast solver for large WAN topologies by leveraging the concentration of traffic demands and capacity. In this chapter, we consider the problem of maximizing the total flow across all demands (Equation 2.2).

Terms	Definitions
$\mathcal{V}_{\text{agg}}, \mathcal{E}_{\text{agg}}, \mathcal{D}_{\text{agg}}, \mathcal{P}_{\text{agg}}$	Nodes, edges, commodities, and paths in the aggregated graph
$\mathcal{V}_x, \mathcal{E}_x, \mathcal{D}_x, \mathcal{P}_x$	Subscript denotes entities in the restricted graph for cluster x
x, η	Each cluster x is a strongly connected set of nodes and η is the number of clusters
$k, K_{xy}, K_{sy}, K_{xt}$	An actual commodity k; the rest are bundled commodities from one source (s) or all nodes in a cluster (x) to a target (t) or to all nodes in a cluster (y)

Table 3.1: Additional notation specific to NCFlow.

3.3 NCFlow

In this section, we describe NCFlow. Our steps are as shown in Figure 3.2. Offline, based on historical demands, we divide the network into clusters and determine paths. Further details are in §3.3.4. Online, we allocate flow to the current demands by solving a carefully constructed set of simpler sub-problems, some of which can be solved independently and in parallel. We describe these sub-problems in §3.3.1. Although they can be solved quickly, disagreements between independent solutions can lead to infeasible allocations; we present a simple heuristic in §3.3.2 that provably leads to feasible flow allocations. In §3.3.3, we discuss extensions that increase the total flow allocated by NCFlow. We also show sufficient conditions under which NCFlow is optimal and matches the flow allocated by MaxFlow. Finally, in §3.3.5, we discuss how NCFlow uses fewer forwarding entries by reusing pathlets within clusters and splitting rules for different demands.

3.3.1 Basic Flow Allocation

We begin by describing a simple (but incomplete) version of NCFlow's flow allocation algorithm; the pseudocode is in Figure 3.4. We continue using Figure 3.3 as a running example. The basic algorithm proceeds in four steps.

In the first step, we allocate flow on the aggregated graph; as shown in MaxAggFlow in Figure 3.4. In the aggregated graph, an example of which is in Figure 3.3 (right), nodes are clusters and the edges are bundled edges from the original graph—the edge between the red and yellow clusters corresponds to the five edges between these clusters on the

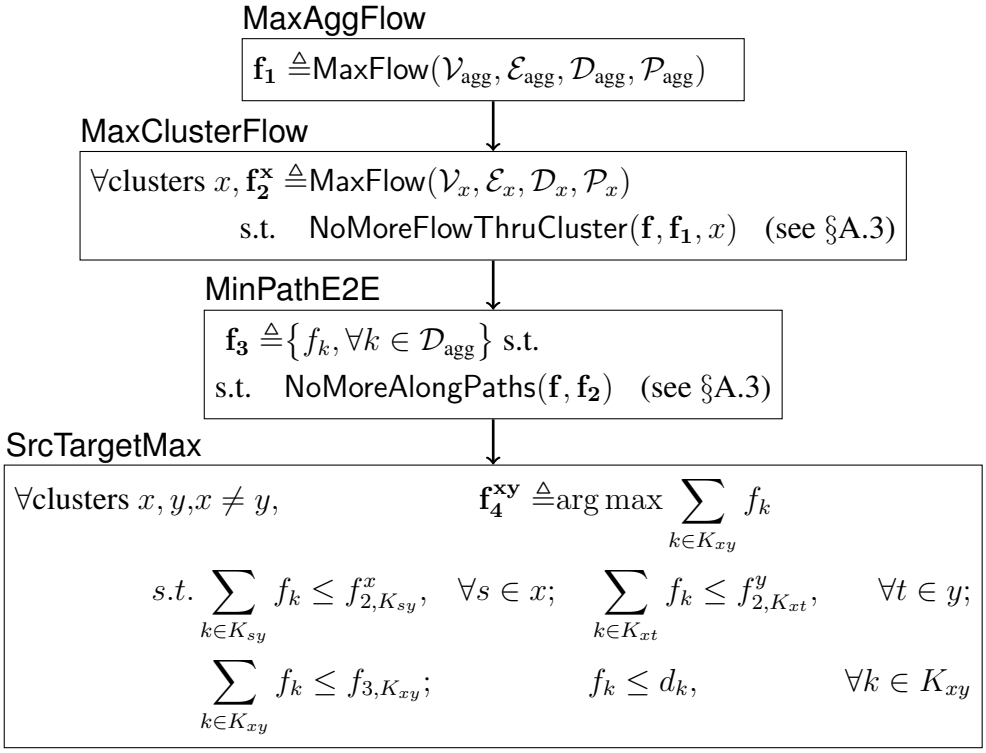


Figure 3.4: The basic flow allocation algorithm used by NCFlow; notation used here is defined in Table 3.1.

actual graph. Similarly, we bundle demands on the aggregated graph: the demand K_{xy} between the clusters x and y corresponds to all of the demands whose sources are in cluster x and targets are in cluster y. The resulting flow allocation ($\mathbf{f_1}$) accounts for bottlenecks on the edges between clusters. However, this flow may not be feasible, since there may be bottlenecks *within* the clusters.

In the second step, we refine the allocation from step 1 to account for intra-cluster demands and constraints. Specifically, we allocate flow for the demands whose sources and targets are within the cluster. We also allocate no more flow than was allocated in $\mathbf{f_1}$ for the inter-cluster flows. MaxClusterFlow in Figure 3.4 shows code for this step. We note a few details:

- We use virtual nodes to act as the sources and targets for the inter-cluster flows; the flow allocated in $\mathbf{f_1}$ determines which virtual node (i.e., which neighboring cluster) is the sender or the receiver for an inter-cluster demand.

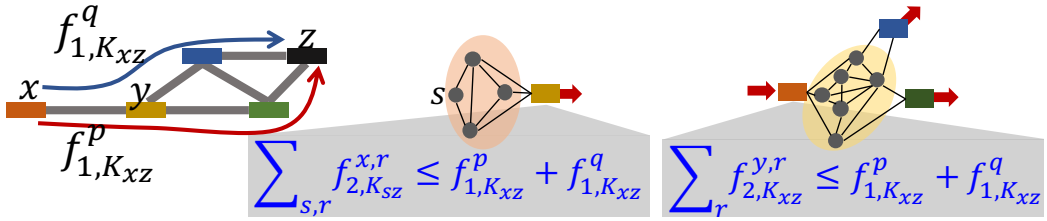


Figure 3.5: An example illustrating how the flow allocated in MaxAggFlow translates to constraints on the flow to be allocated in MaxClusterFlow at two different clusters.

- Figure 3.5 shows two examples on the right where the virtual nodes are drawn using squares.
- Figure 3.5 also shows the NoMoreFlowThruCluster constraints for demands from sources in the red cluster to targets in the black cluster (depicted as x and z respectively). On the aggregated graph, the flow for this demand takes the two paths shown. In the red cluster, as shown in the equation, the traffic from all sources (s), along multiple paths (r) to the virtual node, is restricted to be no more than what was allocated in $\mathbf{f_1}$.
- Figure 3.5 on the right also shows a more complex case that happens in the yellow cluster. Here, the traffic arrives at one virtual node but can leave to multiple virtual nodes. In MaxClusterFlow, we set up paths between all pairs of virtual nodes. As shown in the equation, the traffic leaving the red virtual node on paths (r) to either of the other virtual nodes must be no more than the total flow on paths p and q from $\mathbf{f_1}$.
- Observe that bundling demands ensures fewer variables and constraints for MaxClusterFlow. The demand from red to black clusters comprises twenty node pairs in the actual graph in Figure 3.3 (left); four sources in the red cluster and five targets in the black cluster. However, the MaxClusterFlow for the red cluster only has four bundled demands, from each source to the virtual node, and the yellow cluster has just one bundled demand from and to virtual nodes.

In the third step, we reconcile end-to-end; that is, we find the largest flow that can be carried along each path on the aggregate graph. As shown by MinPathE2E in Figure 3.4, for each bundle of demands and each path, we take the minimum flow allocated ($\mathbf{f^x_2}$) at each cluster on the path.

Problem	# of Nodes	# of Edges	# of Commodities
MaxFlow	N	M	K
MaxAggFlow	η	$\leq \min(M, \eta^2)$	$\leq \min(K, \eta^2)$
MaxClusterFlow	$\sim \frac{N}{\eta} + \eta$	$\sim \frac{M}{\eta} + 2\eta$	$\sim \frac{K}{\eta^2} + 2\frac{N}{\eta} + \eta^2$

Table 3.2: Sizes of the problems in Figure 3.4 using notation from Tables 2.1 and 3.1. Just verifying that flow is feasible (i.e., FeasibleFlow in Equation 2.1) uses $O(K + M)$ number of equations and $O(K)$ variables. NCFlow has one instance of MaxAggFlow and executes the η instances of MaxClusterFlow in parallel. MinPathE2E and SrcTargetMax are relatively insignificant.

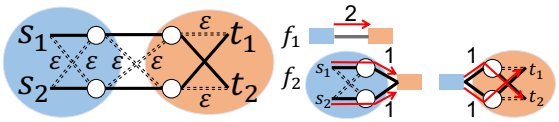
The flow allocation for the demands in a cluster x can be read directly from the $\mathbf{f_2^x}$ solution of MaxClusterFlow. For demands that span clusters, however, more work remains because the steps thus far do not directly compute their flow. In particular, $\mathbf{f_3}$ allocates flow for cluster bundles, such as flows for all demands whose sources are in cluster x and whose targets are in cluster y. The corresponding per-cluster flow allocations, $\mathbf{f_2^x}$ and $\mathbf{f_2^y}$, allocate flow from a source node and to a given target, respectively. Thus, in the final step, SrcTargetMax, we assign the maximal flow to each inter-cluster demand that respects all previous allocations.

Properties of Basic Flow Allocation

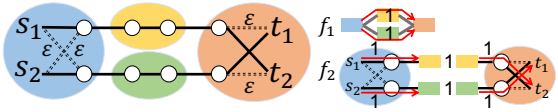
Solver runtime: The numbers of equations and variables in the sub-problems are shown in Table 3.2. If the number of clusters η is 1, note that there is exactly one per-cluster problem, MaxClusterFlow, which matches the original problem from Eqn. 2.2. When using a few tens of clusters, we will show in §3.5 that all of the sub-problems are substantially smaller than the original problem (MaxFlow).

Feasibility: The flow allocated by Figure 3.4 satisfies demand and capacity constraints; we will prove this formally in §A.1.1. For demands whose source and target are in different clusters, however, disagreements may ensue since the different problem instances assign flow to different bundles of edges and demands. We illustrate two such examples in Figure 3.6; both have 1 unit of demand from s_1 to t_1 and from s_2 to t_2. The dashed edges have a capacity of $\varepsilon \ll 1$ and all of the other edges have a very large capacity.

- The example in Figure 3.6a illustrates an issue with bundling edges. The actual graph

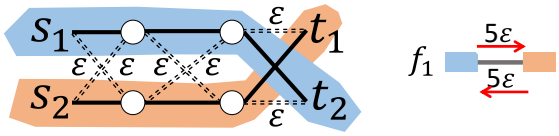


(a) Disagreement arising from bundling edges: As shown on the right, the basic flow allocation algorithm in Figure 3.4 will compute a flow of 2 units, but only 4ε units of flow can be carried; see §3.3.1.

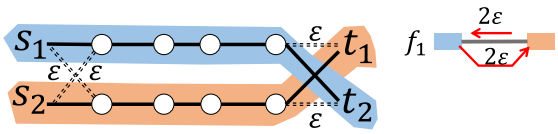


(b) Disagreement arising from bundling demands: As shown on the right, the basic flow allocation algorithm in Figure 3.4 will again compute a flow of 2 units, but only 4ε units of flow can be carried.

Figure 3.6: Two examples that illustrate how disagreements in flow allocation can occur in NCFlow's basic flow allocation algorithm.



(a) For the disagreement problem in Figure 3.6a, a different clustering choice that does not lead to such a disagreement.



(b) For the disagreement problem in Figure 3.4, a different clustering choice that does not lead to such a disagreement.

Figure 3.7: Impact of different clustering choices in NCFlow.

on the left can only carry 5ε units of flow for each demand. However, as the figures on the right show, MaxAggFlow allocates two units of flow since the four edges between these two clusters can together carry all of the two units of demand. The MaxClusterFlow instances also allocate two units of flow as shown. The discrepancy arises because the problems in Figure 3.4 do not know that the *top* egress of the left cluster can take in all of the demand of s_1 but has only a low capacity to t_1.

- The example in Figure 3.6b illustrates an issue with bundling demands. Here too, observing the actual network on the left will show that 2ε units can be carried for each demand split evenly between the top and the bottom path. Again, as the figures on the right show, the basic flow allocation algorithm will conclude that both units of demand can be carried. Here, the discrepancy arises from the bundling of demands, the problems in Figure 3.4 cannot discern that the MaxClusterFlow instance of the left cluster sends the first demand to the brown cluster while the MaxClusterFlow of the right cluster wants to receive the second demand from the brown cluster.

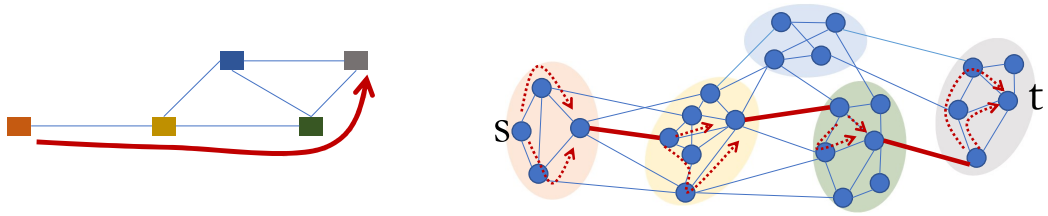


Figure 3.8: To guarantee feasibility, each cluster bundle is allocated flow on only one path on the aggregated graph (left) and on only one edge between each pair of clusters (right); the usable path and edges are shown in dark red. Note that multiple paths can still be used within clusters.

3.3.2 A feasible heuristic

To avoid end-to-end disagreements, we make two simple changes to the basic flow allocation algorithm in §3.3.1.

First, when solving MaxAggFlow, only one path on the aggregated graph can be used for all of the demands between a given pair of clusters; we call such groups of demands to be cluster bundles. Next, between a pair of connected clusters, only one edge can carry the flow for a cluster bundle. Figure 3.8 shows in dark red an example path for a cluster bundle and the allowed edges between clusters; we also show the intra-cluster paths that can carry flow for this bundle.

There are multiple ways to avoid disagreements while keeping the problem sizes small via bundling. We discuss the above changes here because they are simple and sufficient. Specifically, we show that:

Theorem 1. *The algorithm in Figure 3.4, when constrained as discussed above, will always output a feasible flow.*

Proof. The proof is in §A.1.2. Intuitively, these changes suffice because the independent decisions made by different problems in Figure 3.4 cannot disagree; per cluster bundle, all problem instances allocate flow to the same edge and path. □

3.3.3 Stepping towards optimality

The flow allocation algorithm described thus far is fast but not optimal; that is, it may allocate less total flow over all demands than the flow allocated by solving the larger global

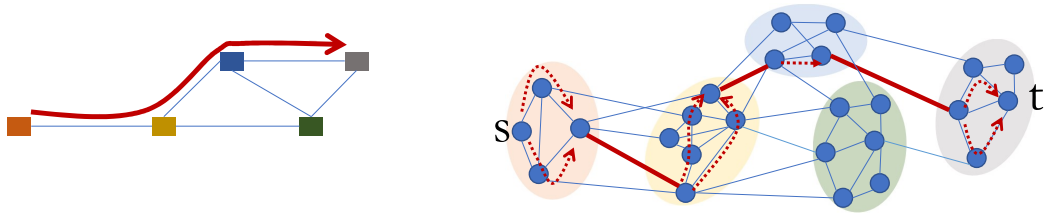


Figure 3.9: In contrast with Figure 3.8, NCFlow produces a different flow allocation in the second iteration of the algorithm. For the same cluster bundle, NCFlow chooses a different path on the aggregate graph and also chooses different inter-cluster edges. The chosen paths and edges are again shown in red.

problem (MaxFlow from Eqn. 2.2). There are a few reasons why this happens. The MaxAggFlow in Figure 3.4 allocates flow on paths through clusters without knowing how much flow the clusters can carry. Switching the order, i.e., solving MaxClusterFlow before MaxAggFlow, could be worse because each cluster must allocate flow without knowing how much flow can be carried end-to-end. Furthermore, the heuristic in §3.3.2 constrains each cluster bundle to use only one edge between clusters and one path on the aggregated graph. We now discuss a few extensions to increase the flow allocation.

First, we re-solve the problems in Figure 3.4 multiple times. A simple way to do this would be to deduct the allocated flow and use the residual capacity on edges in the next iteration. Also, we pick different edges between clusters and/or different paths on the aggregated graph in different iterations (see Figure 3.9 for an example). The number of iterations is configurable; we continue as long as the total flow increases in each iteration by at least a pre-specified amount (say 5%). One could apply other policies such as a timeout. We show in §3.5 that a small number of iterations suffice for a sizable increase in the total flow. We will also show that later iterations finish faster than the first iteration, perhaps because there are fewer demands remaining to satisfy.

Next, we empirically observe that the choice of clusters and edges/paths to use in different iterations has an effect on flow allocation. For instance, the disagreements in Figure 3.6 *go away* by using a different choice of clusters—specifically, see Figure 3.7a and Figure 3.7b. We discuss how NCFlow precomputes cluster and edge/path choice in §3.3.4.

To sum up, we prove that flow allocation will be optimal when a few sufficient conditions hold:

Theorem 2. *The method in Figure 3.4 leads to the optimal flow allocation when any path can be used within each optimization and the number of clusters is* 1*; or the number of clusters is equal to the number of nodes; or all of the following conditions hold:*

- *The aggregated graph $\mathcal{G}_{agg}$ is a tree.*
- *Only one edge connects any pair of clusters.*
- *All demands are satisfiable.*

Proof. By optimal, we mean that the total allocated flow must be as large as an instance of Equation A.4 wherein any path can be used. The proof is in §A.1.3. Intuitively, when the number of clusters is 1 and any paths can be used, a single instance of MaxClusterFlow is identical to the optimal problem in Equation A.4. Similarly, when the number of clusters equals the number of nodes, MaxAggFlow is identical to the optimal problem. Furthermore, the conditions listed lead to optimality because the optimal flow allocation can be transformed into an allocation that can be outputted by Figure 3.4. □

Even though the listed conditions appear restrictive, note that the topology within clusters can be arbitrary. We will show in §3.5 that NCFlow offers nearly optimal flow allocations even when the above conditions do not hold.

3.3.4 Choosing clusters and paths

The choice of clusters and paths affects both the solution quality and runtime of NCFlow. We cast cluster choice as a graph partitioning problem [33, 104, 137] with these objectives:

- **Concentrated with a low cut:** NCFlow can output better flow allocations when much of the total demand and the total edge capacity is between nodes in the same cluster.
- **Balanced cut:** Intuitively, NCFlow will have a smaller runtime when the complexity of MaxAggFlow balances with that of MaxClusterFlow. Recall from Table 3.2 that the former depends on the number of clusters whereas the latter depends on the size of the largest cluster.

We empirically observe, based on experiments with many WANs and different types of demands, that:

- On a graph with N nodes, about $\sqrt{N}$ clusters, irrespective of the clustering technique, leads to the best result, i.e., smallest runtime and fewest forwarding entries while allocating nearly the largest amount of flow possible; see Figure 3.12.
- When choosing the same number of clusters, one of the three considered clustering techniques (described below) generally performs better than the others but not in all cases; see Figure A.4.

Thus, the *optimal* clustering choice for a WAN is unclear; it is possible that hand-tuning or using a learning technique may lead to better-performing clusters. Nevertheless, any of the three simple clustering schemes discussed below already suffice for NCFlow to improve substantially over baselines.

We consider the following clustering choices because they are simple and fast; unless otherwise noted, results in this paper use FMPartitioning.

- FMPartitioning [26,42] divides nodes into clusters so as to maximize a "modularity" score which prefers more edges to lie within than between clusters. In NCFlow, we apply modularity-based clustering with edge weights set to their capacity.
- Spectral clustering [107] computes eigenvectors of the weighted adjacency matrix and chooses a desired number of the top eigenvectors as *cluster heads*; each node is assigned to the cluster of their closest eigenvector (e.g., using k-means).
- Leader Election picks a desired number of nodes at random as leaders and assigns each other node to the closest leader; wherein, distance is measured as the path length using invcap edge weights.

Some other clustering techniques [83, 104, 137] can balance cluster sizes or trade-off between concentration and balance but are more complex computationally; it is possible that using such schemes can further improve NCFlow.

Path choice in NCFlow: On the aggregated graph and on each cluster graph, we precompute offline a small number of paths between every pair of nodes. We consider the following different path choices and pick paths that lead to the largest flow allocation on historical demands:

- k-shortest paths [144] with edge weight of 1 or $\frac{1}{c_e}$ where c_e is the capacity of edge e and $k = 4, 8$ or 16.
- As above, but with the additional requirement that the paths for a node pair are edge-disjoint [105].

NCFlow also pre-computes offline *i)* a pseudo-random choice of which edges to use between a pair of connected clusters in each iteration, and *ii)* which path on the aggregated graph to use for each cluster bundled demand in each iteration.

3.3.5 Setting up switch forwarding entries

NCFlow uses many fewer switch forwarding entries than prior works due to the following reasons.

First, the paths along which NCFlow allocates flow can be thought of as a sequence of pathlets [64, 90, 141] in each cluster connected by crossing edges between clusters. Figures 3.8 and 3.9 illustrate such paths on the right. This observation is crucial because a pathlet can be reused by multiple demands. For example, in Figure 3.8, the flow from any source in the red cluster to any target in the grey cluster would use the same pathlets shown in the yellow, green, and blue clusters. Prior work [71, 72], on the other hand, establishes paths for each demand. Using pathlets has two advantages. The number of pathlets used by NCFlow is about η times less than the number of paths used by prior works.[1] Furthermore, a typical pathlet has fewer hops than a typical end-to-end path. Thus, NCFlow uses many fewer rules to encode paths in switches.

Next, whenever NCFlow allocates flow at the granularity of cluster bundles, all of the demands in a bundle take the same paths and are split in the same way across paths. Hence, NCFlow uses one traffic splitting rule for all demands in such bundles. For instance, the demands from source s in the red cluster in Figure 3.8 to any target in the grey cluster are split with the same ratio across the same pathlets in all clusters (except the grey cluster where they take different pathlets to reach their different targets). Thus, with NCFlow, the

[1] More precisely, the number reduces from $PN(N-1)$ to $\sum_x P(N_x)(N_x - 1)$ where P is the number of paths per node pair, the N nodes are divided into η clusters, and cluster x has N_x nodes. If clusters are evenly sized, $N_x = N/\eta$, and the ratio of these terms is $\sim \eta$.

number of splitting rules at a source decreases by a factor of $\sqrt{N}/2$.[2]

The paths and splitting rules to push into switch forwarding tables are determined by the offline component of NCFlow and only change occasionally. After each allocation, only the splitting ratios change. More details on the data-plane of NCFlow such as how to compute the total flow that can be sent by each demand and the splitting ratios as well as how to move packets from one pathlet to the next are in section A.2. In §3.5, we measure the numbers of rules used by NCFlow.

3.4 Implementing NCFlow

Our current prototype of NCFlow is about 5K lines of Python code, which invokes Gurobi [68] v8.1.1 to solve all of the optimization problems. For clustering WAN topologies, we adapt [43] to find clusters that maximize modularity; we also use our own implementation of NJW spectral clustering [107]. We use a grid search over the number of clusters (η) and the above clustering techniques to identify the best performing choice for each topology on a set of historical traffic matrices. To compare with state-of-the-art techniques, we customize the public implementations of SMORE [86, 87] and TEAVAR [27]. We have also implemented Fleischer's algorithm [53]; our implementation is about $10\times$ faster than public implementations [74][3] since we carefully optimize a key bottleneck in Fleischer's algorithm. All of these code artefacts are available on GitHub at https://github.com/stanford-futuredata/pop-ncflow.

3.5 Evaluation

We evaluate NCFlow on several WAN topologies, traffic matrices, and failure scenarios to answer the following questions:

- Compared to state-of-the-art LP solvers and approximate combinatorial algorithms, does NCFlow offer a good trade-off between runtime and total flow allocation? Is it

[2] A source uses $N-1$ splitting rules in prior works but with NCFlow only requires $N_x+\eta-2$ rules when the source's cluster has N_x nodes; if clusters are evenly sized and $\eta \sim \sqrt{N}$, the ratio of these terms is $\sqrt{N}/2$.

[3] https://github.com/eigenpi/mcf_solver

Topology	# Nodes	# Edges	# Clusters
PrivateLarge	$\sim$ 1000s	$\sim$ 1000s	31
Kdl	754	1790	81
PrivateSmall	$\sim$ 100s	$\sim$ 100s	42
Cogentco	197	486	42
UsCarrier	158	378	36
Colt	153	354	36
GtsCe	149	386	36
TataNld	145	372	36
DialtelecomCz	138	302	33
Ion	125	292	33
Deltacom	113	322	30
Interoute	110	294	20
Uninett2010	74	202	24

Table 3.3: Some of the WAN topologies used in our evaluation; see §3.5.1. The networks in blue are publicly available topologies from the Internet Topology Zoo [82]; they can be found at http://www.topology-zoo.org/.

substantially faster, with only a small decrease in total flow?

- For real-world TE scenarios, in which flow solvers must adapt to changing demands and faults, how much benefit does NCFlow offer relative to the state-of-art?
- How do our various design choices in NCFlow impact its performance?

3.5.1 Methodology

Here, we describe our methodology—the topologies, traffic, baselines, and metrics used in our evaluation.

Topologies: We use two real topologies from a large enterprise—PrivateSmall is a production internet-facing WAN with hundreds of sites, and PrivateLarge is a larger WAN that contains many more sites. We also use several topologies from the Internet Topology Zoo [82] and reuse topologies used by prior works [27, 76]. Table 3.3 shows details for some of the used topologies; note that the topologies shown are $10\times$ to $100\times$ larger than those considered by prior work [27, 71, 76, 86, 96].

Traffic Matrices (TMs): We benchmark NCFlow on traffic traces from PrivateSmall,

which contain the total traffic between node pairs at 5-minute intervals. We also generate the following kinds of synthetic traffic matrices for all topologies:

- **Poisson**$\big(\lambda, \delta\big)$ models demands with varying concentration; the demand between nodes s and t is a Poisson random variable with mean $\lambda\delta^{\mathrm{d}_{st}}$, where d_{st} is the hop length of the shortest path between s and t and $\delta \in [0, 1)$ is a *decay factor*. We choose δ close to 0 or to 1 to model strongly and weakly concentrated demands, respectively.
- **Gravity**$\big(v\big)$ [18, 123]: The total traffic leaving a node is proportional to the total capacity on the node's outgoing links (parameterized by v); this traffic is divided among other nodes proportional to the total capacity on their incoming links.
- **Uniform**$\big([0, a)\big)$: The traffic between any pair of nodes is chosen uniformly at random, between 0 and a.
- **Bimodal**$\big([0, a), [b, c), p\big)$ [18]: A p fraction of the node pairs, chosen uniformly at random, receive demands from **Uniform**$\big([b, c)\big)$ while the rest receive demands from **Uniform**$\big([0, a)\big)$. We use $p = 0.2$.

For each above model, we select parameters such that fully satisfying the traffic matrix leads to a maximum link utilization of about 10% in each topology. Then, we scale all entries in the TM by a constant $\alpha \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. Doing so creates demands that range from easily satisfiable to only partially satisfiable; with $\alpha = 128$, the satisfiable portion of the demand varies between 25-70%. We generate five samples for each traffic model and scale factor for each topology.

Baselines: We compare NCFlow with these techniques:

Path Formulation (PF_4) solves the multi-commodity max-flow problem shown in Equation 2.2 using k-shortest paths between node pairs where $k = 4$. Results for other path choices are in §A.6.4.

PF *Warm Start* ($\mathsf{PF}_{4\mathsf{w}}$) matches PF_4 except that it allows the LP solver to "warm start"; that is, over a sequence of traffic matrices, the flow allocated to the previous TM is used as a starting point to compute allocation for the next TM. When traffic changes are small, warm start leads to faster solutions.

CSPF: We implement Constrained Shortest Path First [58], a greedy heuristic that we previously described in §2.3.

Approximate Combinatorial Algorithms: Fleischer's algorithm [53] is the best-known approximation for MaxFlow. We use two variants: Fleischer-Path where flow is restricted to a path set and Fleischer-Edge without any path restrictions. We show results here for an approximation guarantee of 0.5; that is, the techniques must achieve at least half of the optimal flow allocation.

SMORE [86] allocates flow dynamically on paths that are pre-computed using Räcke's Randomized Routing Trees (RRTs). We use the code from [87] to compute paths. Since the LP in [87] requires demands to be fully satisfiable, we implement a variant, SMORE*, that maximizes the total flow on the computed paths, regardless of demand satisfiability.

TEAVAR [27] models link failure probabilities and computes flow allocations given an availability target.[4] We implement a variant, TEAVAR*, that maximizes the total flow[5]; further details are in section A.5.

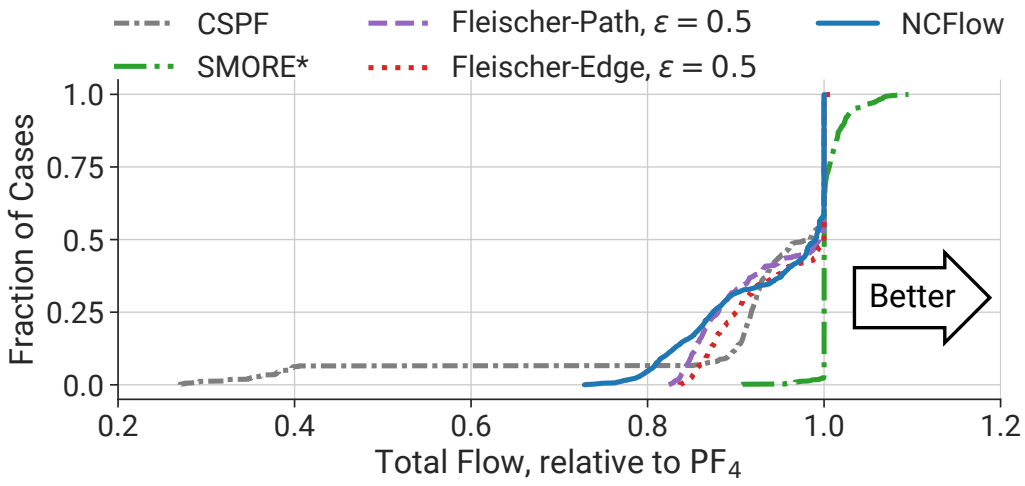
Clusters, Paths, and # of Iterations: Table 3.3 shows the number of clusters used by NCFlow per topology. Here, we report results on edge-disjoint paths, chosen using inverse capacity as the edge length; results for other path choices are qualitatively similar (see §A.6.4). All schemes that use paths (i.e., PF_4, Fleischer-Path, TEAVAR*, and NCFlow) use the same method to compute paths. For each iteration up to $\mathcal{I} = 6$, we also pre-compute offline the path to use on the aggregated graph, and the edge to use between connected clusters for each cluster bundle.

Metrics: We compare the schemes on the following metrics:

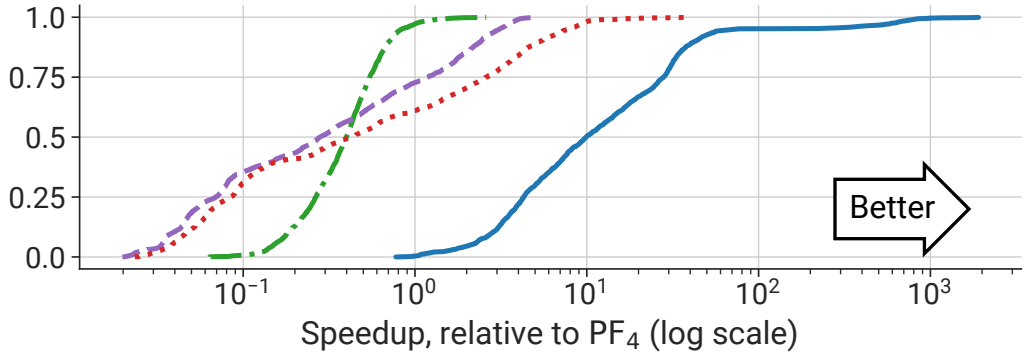
- **Relative total flow** is the total flow achieved by a scheme relative to PF_4.
- **Speedup ratio** is the runtime of each scheme relative to PF_4. For LP-based methods, we report the Gurobi solver runtimes, since models can be constructed once offline in practice. For combinatorial methods, we report algorithm execution time. All runtimes are measured on an Intel Xeon 2.3GHz CPU (E52673v4) with 16 cores and

[4] The open-source code for TEAVAR can be found at https://github.com/manyaghobadi/teavar.

[5] TEAVAR solves the Maximum Concurrent Flow objective; see Table 2.2.



(a) CDF of total flow relative to PF_4



(b) CDF of speedup relative to PF_4

Figure 3.10: CDFs comparing NCFlow with state-of-the-art methods. With only a modest decrease in total flow, NCFlow offers a substantial runtime speedup.

112 GB of RAM.

- **FIB Entries:** We measure the number of switch forwarding entries used.

3.5.2 Comparing NCFlow to the State of the Art

Figures 3.10a and 3.10b show cumulative density functions (CDFs) of the relative total flow and speedup ratio for NCFlow and several baselines. These results consist of $2,600$ traffic matrices and 13 topologies. If a scheme matches the baseline PF_4, its CDF will be a

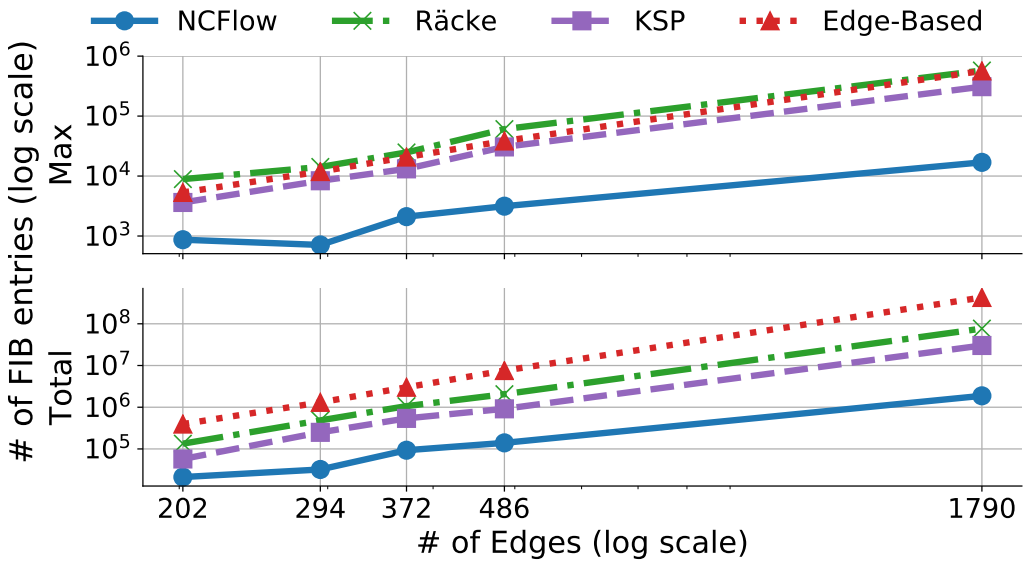


Figure 3.11: Comparing the number of forwarding entries used by various methods for the experiments from Figure 3.10.

pulse at $x = 1$ in both figures; the fraction of cases to the left (or right) of $x = 1$ indicate how often a scheme is worse (or better) than PF_4. Note that the x-axis for the speedup ratio is in log scale.

We see that SMORE*, shown using green dashed lines in the figures, modestly improves the flow allocation (in 25% of the cases) while almost always taking longer to run than PF_4. Both effects are because SMORE* allocates flow on Räcke's RRTs instead of k-shortest paths.

The edge and path variants of Fleischer's, shown using purple and red lines in the figures, perform similarly; since they are approximate algorithms, they allocate less flow than PF_4 in roughly 50% of cases, but are also faster than PF_4 in slightly less than 50% of cases. We conclude that these approximate algorithms are not practically better than PF_4.

In contrast, NCFlow, shown with dark blue lines in the figures, almost always allocates at least 80% of PF_4's total flow, while achieving large speedups. In the median case, NCFlow achieves 98% of the flow and is over $8\times$ faster. These improvements accrue from NCFlow solving smaller optimization problems than PF_4.

Figures A.2 and A.3 tease apart the above results by load, traffic type and topology. Figures A.6–A.9 show results for alternate path choices. Taken together, these results indicate

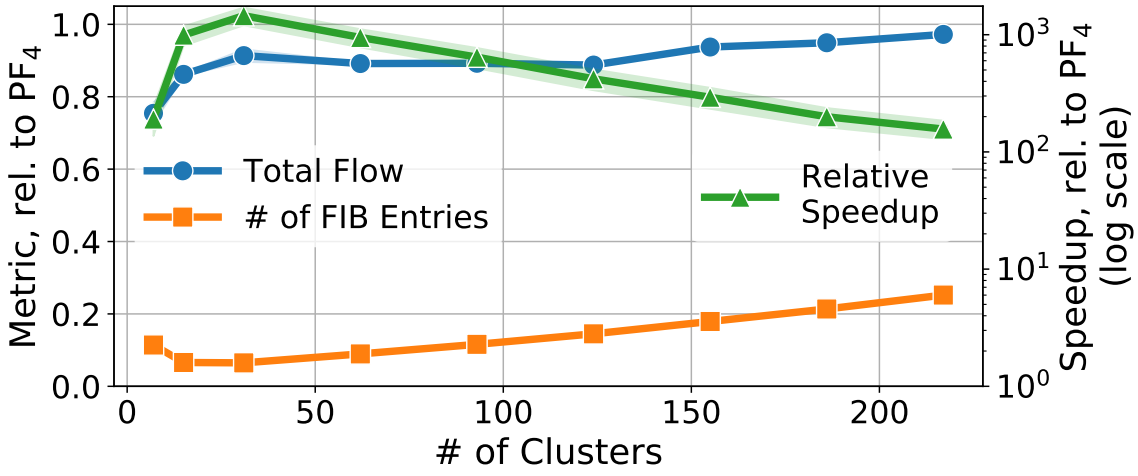


Figure 3.12: NCFlow's performance when using different numbers of clusters on PrivateLarge. The speedup ratio is plotted on the right y-axis in log scale; the other metrics use the left y-axis.

that NCFlow's improvements hold across a variety of scenarios.

For the same experiments considered above, Figure 3.11 shows the number of switch forwarding entries used in different topologies. (A full set of results is in Table A.1.) The bottom plot is the total number of forwarding entries across all switches, while the top shows the maximum for any switch. Note that both the x and y axes are in log scale. NCFlow consistently uses fewer forwarding entries; using NCFlow offers a greater amount of relative savings than switching from all edges to just a handful of paths per commodity. The savings from NCFlow also increase with topology size. The reason, as noted in §3.3.5, is that NCFlow reuses pathlets and traffic splitting rules for many different commodity.

3.5.3 Effect of Design Choices

Figure 3.12 shows how NCFlow's performance varies with the numbers of clusters used on PrivateLarge. While NCFlow allocates roughly the same amount of total flow, using about 30 clusters improves runtime and reduces forwarding entries. Figure A.4 compares NCFlow's performance when using different clustering techniques; more details are in §A.6.2.

Recall from §3.3.3 that NCFlow uses multiple iterations of Figure 3.4. In the above experiments, the first iteration alone accounts for 75% of the runtime and for roughly 90%

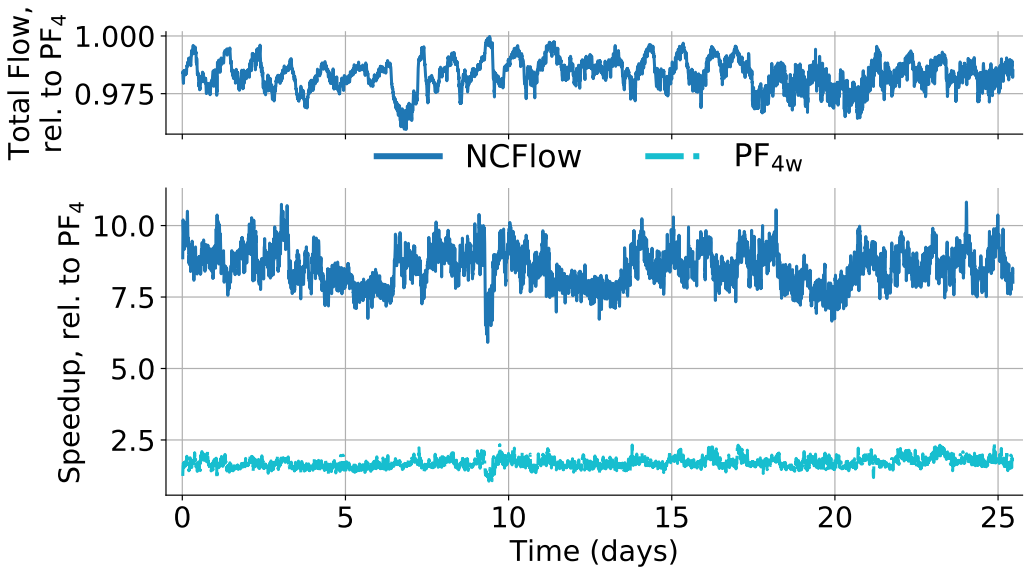


Figure 3.13: Allocated flow and speedup relative to PF_4 on a sequence of production TMs from PrivateSmall. In half of the cases, NCFlow allocates at least 98.5% of the flow and is at least $8.5\times$ faster.

of the flow that is allocated by NCFlow. Later iterations are faster perhaps because they have less traffic to consider.

Breaking down the runtime by the steps in Figure 3.4, we see cases where MaxClusterFlow accounts for over 70% of NCFlow's runtime perhaps because the largest cluster contains a large fraction of the nodes. Better cluster choice or recursively dividing the largest clusters can further lower runtime.

3.5.4 NCFlow on Real-World Traffic

Here, we experiment with a sequence of traffic traces collected on the PrivateSmall WAN. Figure 3.13 plots the moving average (over 5 windows) of the total flow and speedup relative to PF_4 for two schemes—NCFlow in blue and $\mathsf{PF}_{4\mathsf{w}}$ in light blue. The figure shows that $\mathsf{PF}_{4\mathsf{w}}$'s warm start yields a median speedup of $1.66\times$. NCFlow achieves a consistently higher speedup ($8.5\times$ in the median case), and the flow allocation is nearly optimal: the median total relative flow is 98.5%, and NCFlow always allocates more than 93%.

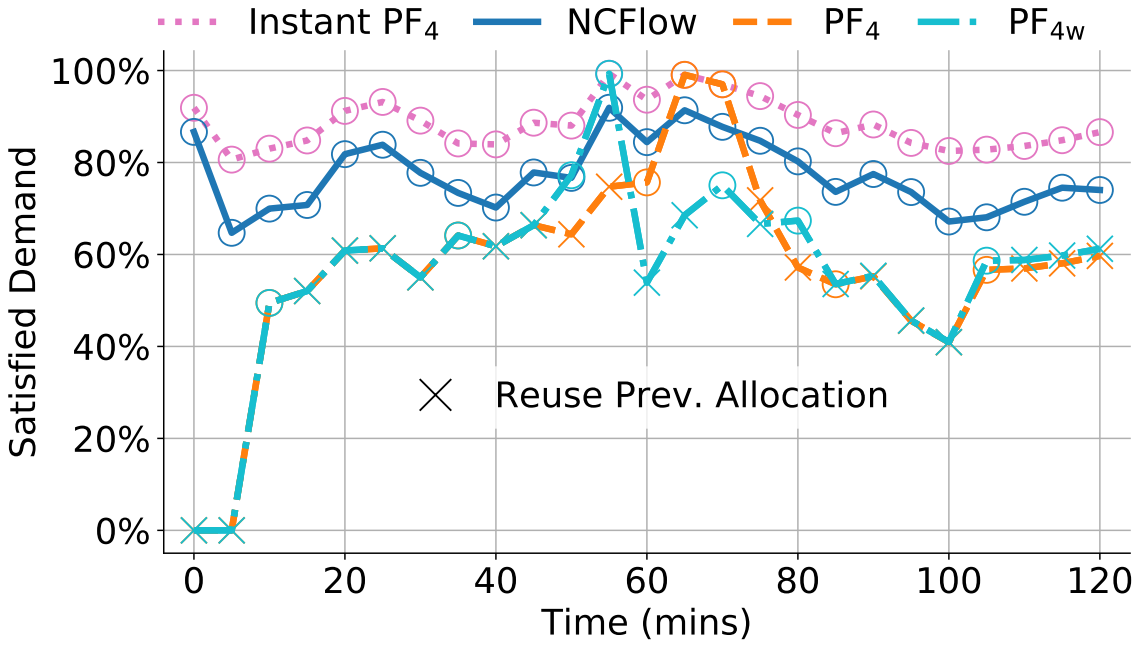


Figure 3.14: When demands change, how solver runtimes affect flow allocation on PrivateLarge: Due to the slow runtime, PF_4 and PF_{4w} carry only 62% of the traffic that can be satisfied by Instant PF_4, a (hypothetical) scheme which has zero runtime. NCFlow carries 87% of the traffic since its faster runtime compensates for its sub-optimality.

3.5.5 Tracking Changing Demands

Here, we evaluate the impact of a technique's runtime on its ability to stay on track with changing demands. Specifically, on the PrivateLarge topology, we use a time-series of traffic matrices, wherein a new TM arrives every five minutes and the change from one TM to the next is consistent with the findings in Figure 2.3. At each time-step, all techniques have the opportunity to compute a new allocation for the current TM or to continue computing the allocation for an earlier TM if they have not yet finished; in the latter case, their most recently computed allocation will be used for the current TM. For example, a technique that requires five minutes to compute a new allocation will be always *one window behind*, i.e., each TM will receive the allocation that was computed for the previous TM.

Figure 3.14 shows the fraction of demand that is satisfied by three different schemes; we also show the value for an instantaneous scheme which is not penalized for its runtime. PF_4's average runtime here is over 15 minutes; hence, as the orange dashed line shows,

PF_4 is able to compute a new allocation only for every third or fourth TM. This leads to substantial demand being unsatisfied: for node pairs whose current demand is larger than before, PF_4 will not allocate enough flow. On the other hand, node pairs whose current demand is less than their earlier demand will be unable to fully use PF_4's allocation. As the figure shows, PF_4 only satisfies 53% of the changing demand on average, whereas Instant PF_4 satisfies 87% of the demand.

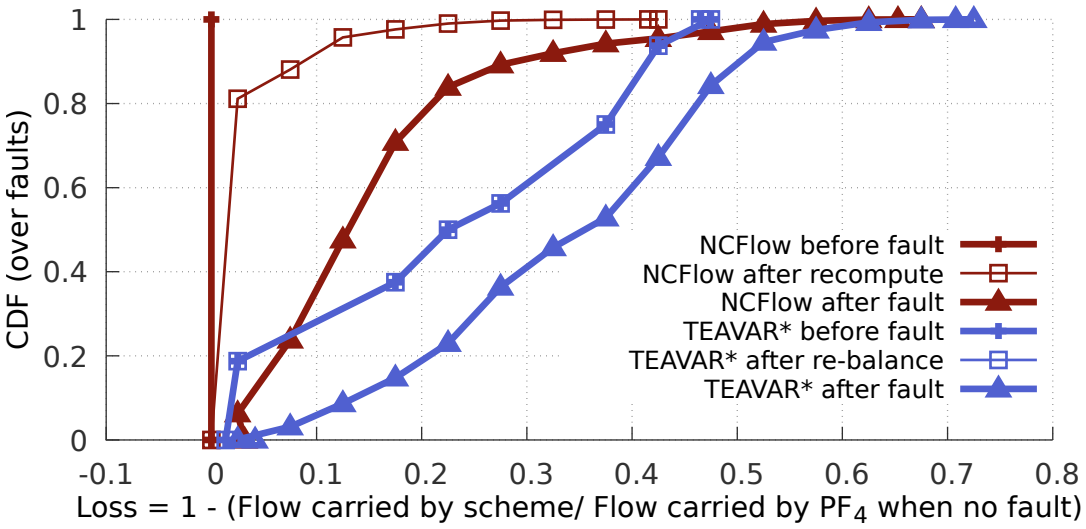
$\mathsf{PF}_{4\mathsf{w}}$ (the dash-dot light blue line), where the solver warm starts using the previous allocation, is modestly faster than PF_4 on average. As the figure shows, the average demand satisfied by $\mathsf{PF}_{4\mathsf{w}}$ is only slightly larger than PF_4 (about 54%).

In contrast, NCFlow (the solid dark blue line) finishes well within five minutes which allows allocations to change along with the changing demands. We find that on average NCFlow satisfies 75% of the demands; its smaller runtime more than makes up for sub-optimality, allowing NCFlow to carry more flow than PF_4 when demands change.

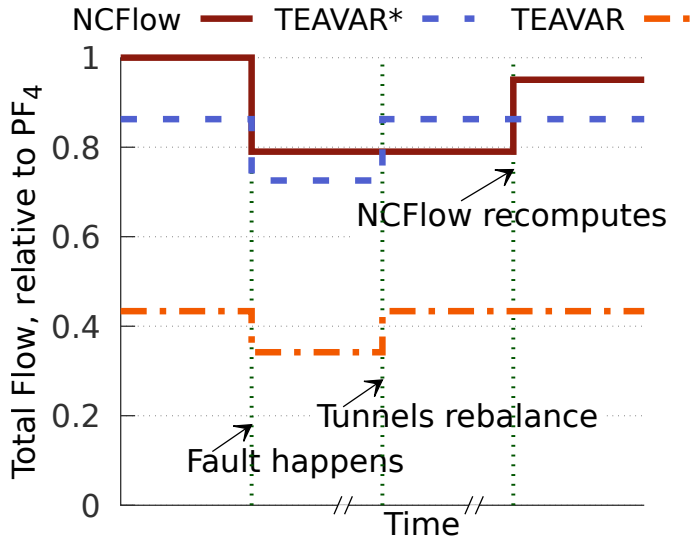
3.5.6 Handling Failures with NCFlow

Here, we evaluate the effect of link failures. As we note in §A.5, TEAVAR* did not finish within several days on any of the topologies listed in Table 3.3 because when all possible 2-link failure scenarios are considered, the number of equations and variables in the optimization problem increase from $O(N^2)$ for MaxFlow to $O(M^2N^2)$ for TEAVAR [27], where N and M are the numbers of nodes and edges, respectively. Hence, we report results on the 12-node, 38-edge WAN topology from B4 [76]. We generate synthetic traffic matrices as noted in §3.5.1. Using link failure probabilities from the open-source TEAVAR implementation, we generate several hundred failure scenarios and, for each TM, we measure the flow carried by NCFlow and TEAVAR* before the fault, immediately after the fault, and after recovery.

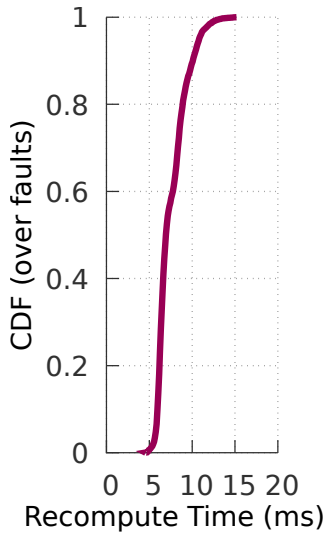
A key difference in fault recovery between NCFlow and TEAVAR* is that TEAVAR* requires sources to rebalance the traffic splits when a failure happens; doing so takes about one RTT on the WAN. Given a parameter β, TEAVAR* guarantees that there will be no flow loss after the tunnels re-balance with a probability of $1-\beta$. See §A.5 for more details. We use $\beta = 0.99$, as recommended in [27]. NCFlow, on the other hand, recomputes flow



(a) CDFs of the flow *loss* before faults, immediately after faults and after recovery (B4 topology, many traffic matrices and faults; see §3.5.6).



(b) Timelapse of when a fault occurs (B4 topology, Uniform traffic matrix, $\beta = 0.99$)



(c) NCFlow's time to recompute after fault.

Figure 3.15: Comparing failure response of NCFlow with prior work.

allocations taking into account the links that have failed; doing so takes one execution of NCFlow and some RTTs to change the traffic splits at switches; more details are in §A.4.

Figure 3.15c shows that the recomputation time is well within one RTT on the WAN.

Figure 3.15b shows a timelapse of the flow carried on the network before the fault, immediately after the fault, and after recovery. As the figure shows, TEAVAR* can have a smaller loss and for a shorter duration; i.e., until sources rebalance traffic while NCFlow can carry more flow before fault and after recovery; moreover, the fast solver time can reduce the duration of loss.

Figure 3.15a shows CDFs over many faults and traffic matrices for NCFlow and TEAVAR*. We record the flow loss at three stages: before the fault, immediately after the fault, and after recovery. As the figure shows, NCFlow's ability to carry more flow before the fault and after recovery more than compensates for the slightly larger loss it may accrue in between.

3.6 Discussion

Extending beyond MaxFlow: FeasibleFlow is a common constraint for many objectives beyond MaxFlow (see Table 2.2). Since the algorithm in §3.3.1 and the heuristic in §3.3.2 guarantee feasibility, NCFlow can apply to objectives beyond MaxFlow; however, we believe that more work is needed to improve the solution quality for different objectives.

Optimality guarantee: In §A.8, we show that constraining by clusters and paths, as done by NCFlow, does not necessarily reduce the flow allocation; that is, nearly the maximum amount of flow can be carried while respecting clustering and path constraints. This is promising because a better heuristic (than Figure 3.4) may allocate more flow without losing the benefits of solving smaller per-cluster problems. Furthermore, although NCFlow achieves sizable speedups by using simple clustering methods, the optimal cluster choice is uncertain; we show examples in §A.7 to illustrate the challenges.

Recursive (or multiple levels of) clusters: For large topologies or when the largest cluster has a disproportionate number of nodes, we can further divide a cluster into sub-clusters. Doing so is an extension of the algorithm in Figure 3.4 where, in the iterative step, the MaxClusterFlow problem at a cluster is replaced with a new instance of all of the steps in Figure 3.4 along with the additional constraints that arise from the current level (e.g., NoMoreFlowThruCluster constraints). We leave further details to future work.

NCFlow is **agnostic to the underlying solver** used for the problems in Figure 3.4 and can benefit from future improvements to LP solvers and approximate methods [53, 59, 80].

Further use cases: Beyond serving as a drop-in replacement for today's production WAN traffic controllers, NCFlow can be used whenever fast and close-to-optimal solutions are desirable such as: when allocating flow for future time-steps [77, 79] or to compare topology changes [22, 37] or to accelerate the training of ML-based routing systems [135].

3.7 Related Work

NCFlow builds upon a few themes in prior work. We discuss and evaluate against some prior works already. To recap:

- Some large enterprises use path-based global optimization problems similar to MaxFlow to manage traffic on their WANs [71, 72, 76]. We saw in §3.5 that doing so does not scale to the WAN topologies of today or the future, which consist of thousands of sites.

- We saw that approximate algorithms for multi-commodity max flow, such as [53], require a large number of switch forwarding entries since they can send flow along any edge. Also, NCFlow allocates more flow and is faster compared to path-based versions of these algorithms.

- Probabilistic fault protection schemes such as TEAVAR [27] take an infeasibly long time to run on large topologies when considering multiple link failures; they also allocate less flow to reserve capacity to deal with possible failures. Other oblivious techniques [17, 18, 27, 86, 96, 139] have a similar trade-off. Quickly recomputing using NCFlow trades off slightly more loss after a fault to carry much more traffic before the fault and after recomputation.

Hence, we believe that NCFlow is better suited to enterprise WANs, which target very high link utilization and have traffic that is elastic to short-term loss (e.g., scavenger-class traffic, such as replicating large datasets [71, 76, 96]). Here, we discuss other related work.

TE on WANs: Typically, a WAN node is not a single switch, but rather a group of switches connected in a specific way such as a full mesh. Similarly, a WAN edge is a systematic collection of links between many switches. [72] discusses how to hide the intra-node connectivity from the global TE solution. NCFlow complements this technique; it can use a similar intra-node scheme and can support WANs that are $10\times$ larger than were considered in [72]. The specific contraction used by NCFlow—node clusters with large capacity and/or demand between themselves—also differs from the contractions used in route planning [1, 11, 23]. Some BGP-based TE schemes [41, 125, 143], which address how best to move traffic between different (BGP) domains, are also complementary to NCFlow which considers the WAN of a single enterprise (domain). Other TE schemes use different protocols, such as OSPF, or work over longer timescales (e.g., hours to days) [56, 77, 89, 101].

Multi-Commodity Flow Solutions: Both the edge- and path-based LP formulations are well-studied [24, 140]. Some prior work considers the case of a single commodity, i.e., one source and target, and does not directly extend to the case of multiple commodities [70, 94, 113]. The best-known approximate algorithms for multi-commodity flow problems incrementally allocate flow on the shortest path and increase the length of all edges on that path [25, 53, 59, 80]. For the problem sizes considered here, LP solvers such as Gurobi are faster in practice, perhaps because they take larger steps towards the optimal allocation. There is also prior work that customizes the LP solver to improve performance on flow problems [38, 97]. NCFlow is agnostic to the solver used; that is, NCFlow can use any fast solver for the sub-problems in Figure 3.4.

Decompositions: Using standard decomposition techniques for large optimization problems, such as Dantzig-Wolfe and Benders [24, 29], for multi-commodity flow problems has led to inconclusive results [61, 111]; i.e., not consistently faster than MaxFlow. NCFlow can be thought of as a problem-specific decomposition that leverages the observation that both capacity and demands are concentrated in today's WANs.

3.8 Summary

In this chapter, we presented NCFlow, a fast and practical solution for traffic engineering on large WANs. We use geographic partitioning and leverage the concentrated nature of demands and topologies to divide nodes into clusters and solve sub-problems per cluster and on the aggregated graph. Our heuristics guarantee feasibility and empirically achieve close-to-optimal flow allocations. By reusing pathlets and splitting rules across demands, we require fewer forwarding entries in switches. Empirically, on topologies that are over $10\times$ larger than were considered in prior work and many traffic matrices, NCFlow is $8.2\times$ faster than the state of the art, while allocating 98.8% of the total flow and using $6\times$ fewer forwarding entries in the median case. We demonstrate that NCFlow offers sizable benefits when tracking changing demands and reacting to failures. As enterprise WANs continue to grow, we believe techniques such as NCFlow can enable improved traffic orchestration and higher utilization.

Chapter 4

POP

4.1 Introduction

In this chapter, we introduce POP, our second technique for traffic engineering based on commodity-based partitioning. Like NCFlow, POP is faster than the state of the art but only sacrifices a small amount of flow. Additionally, POP supports multiple TE objectives, such as maximum total flow, maximum concurrent flow, and min-max link utilization. Because of its performance and generality, we believe that, just like NCFlow, POP is a suitable choice for traffic engineering on large-scale WANs.

To understand the motivation behind POP, we first examine the underlying computation involved in state-of-the-art TE today. As we discussed in Chapter 2, the centralized approach requires solving a mathematical optimization problem, which is typically a linear program. (Refer to Table 2.2 for examples.) Unfortunately, solving these mathematical programs can be computationally expensive: the worst-case complexity for linear programs is approximately $O(n^{2.373})$ [44, 91], where n is the number of problem variables. Even though LPs can sometimes be solved faster, this depends on the problem structure and the numerical solver used. For large-scale WANs, these LPs can have millions of variables (e.g., k variables for every commodity, where k is the number of paths per commodity) and even more constraints. Naturally, this leads to long solution times, which we demonstrated in Figure 1.2. Heuristics offer an alternative, but they often make assumptions too strong for the problems at hand and produce sub-optimal results, as we discussed in §2.3.

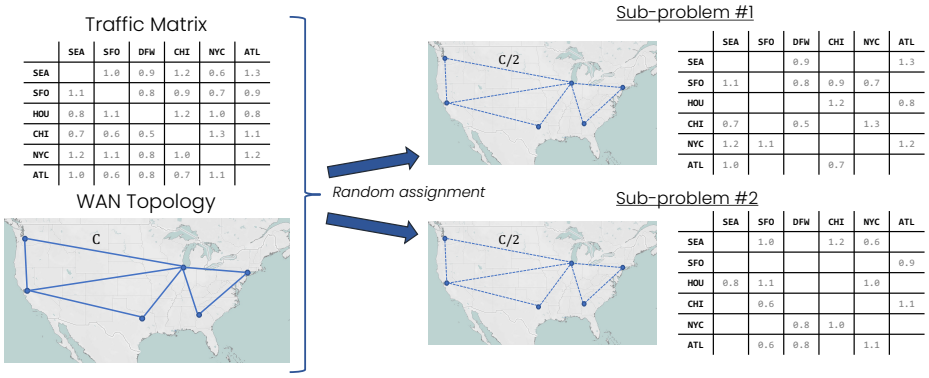


Figure 4.1: Overview of POP's commodity-based partitioning strategy for $l = 2$ sub-problems on a fictitious topology and traffic matrix. The commodities are assigned randomly to one of the sub-problems; the capacities are split evenly.

NCFlow addressed this problem by partitioning the WAN into geographic clusters; because each cluster could now be solved independently, the total number of variables across the resultant set of LPs paled in comparison to the original LP. But the downside of geographic partitioning is that it created a reconciliation problem: disagreements between two or more clusters on inter-cluster commodities could end up with 0 allocated flow, a worst-case scenario that can occur when the traffic matrix overwhelms the WAN topology and links become oversubscribed. (We discuss how this situation can occur in more detail in Chapter 5.)

Instead, we wish to devise an alternative strategy that still partitions the WAN and traffic matrix but *preserves* the underlying topological structure: rather than omit or bundle links due to clustering, we wish to keep the existing topology in place with our partitioning. We also note that the TE problem has other exploitable properties: although the number of commodities and links are both large, each commodity typically requests a small fraction of the total available capacity, and the link capacities are fungible or substitutable—a commodity can make similar progress using *different* links, so long as the links belong to the commodity's set of available paths.

Because of these characteristics, we deem the traffic engineering LP to be a *granular* allocation problem, which we will define in Chapter 4. For such problems, we propose POP, which stands for Partitioned Optimization Problems. To apply POP, we divide the commodities *randomly* among l identical copies of the given WAN topology, each with

a subset of the commodities. The randomized strategy is effective here because of the fungible nature of the problem. Additionally, the link capacities are divided *evenly* among the l sub-problems, so that every link is present across every copy of the topology. We call this approach *commodity-based partitioning*; Figure 4.1 shows an example of how this can be applied on a simple WAN and traffic matrix.

After partitioning, each sub-problem has fewer equations and variables, leading to a super-linear runtime speedup, and we can also execute the sub-problems in parallel. The overall flow allocation is a simple sum of the allocations from the individual sub-problems. Our results show that randomly dividing the commodities and evenly dividing the link capacities among sub-problems works well when commodities are numerous and individually use only a small fraction of the available capacity in the WAN. Empirically, we show that POP's allocations are nearly optimal on several optimization problems, including using real-world inputs.

A supplemental benefit of POP's simplicity is that we can *reuse* the original linear program formulation on each individual sub-problem; unlike NCFlow, we do not have to modify the LP, or design new LPs in its stead. This means that we can support multiple TE objectives—not only maximum total flow, but also maximum concurrent flow and min-max link utilization—all with just a few lines of code.

In the real world, not all traffic matrices precisely fit the definition of granularity that we presented above. For example, a traffic matrix could have "large" commodities with substantial bandwidth demand. Fortunately, in these cases, we can transform the problem into a granular one using *commodity splitting*: the "large" commodities can be split into multiple virtual commodities who each receive partial allocations from multiple sub-problems. Since the number of "large" commodities is small, by definition, POP's sub-problems remain small and still achieve a sizable runtime speedup.

We found that POP is effective on a wide range of topologies and traffic matrices. Using the same evaluation framework previously applied to NCFlow, we observed that POP achieves empirical runtime improvements on the maximum total flow objective of up to $18\times$ in the median case compared to the original problem formulation for maximum total flow, all while staying within 0.1% of optimality. Additionally, POP achieved a $56\times$ speedup with a 1.5% reduction in optimality on the min-max link utilization objective, and

a $2,500\times$ speedup with a 5% reduction in optimality on the maximum concurrent flow objective. Lastly, we found commodity splitting to be a useful optimization when applying POP to "skewed" traffic matrices. Like NCFlow, our implementation is available at the same URL: https://github.com/stanford-futuredata/pop-ncflow.

4.2 Partitioned Optimization Problems for TE

When a traffic engineering problem is granular, we can split it into sub-problems, where each sub-problem has a subset of the commodities from the original problem. Importantly, all links are preserved in each sub-problem; otherwise, this could potentially create scenarios, where a sub-problem attempts to allocate flow on a path, but the link is missing. We leverage the large number of commodities to randomly partition them problems; this procedure yields high-quality flow allocations in expectation due to the Law of Large Numbers. We call this technique Partitioned Optimization Problems (or POP for short). In the rest of this section, we describe the intuition, procedure, and benefits of POP.

4.2.1 Intuition

Optimization problems for large-scale traffic engineering take a long time to solve in part because they have many variables. For example, consider a WAN that has N nodes and M links. Assuming that every site has traffic to send to every other site (i.e., the traffic matrix is relatively full), this gives us N^2 commodities. If we choose to select flows from k different paths on each commodity, then we have a matrix of N^2k variables. Additionally, we would have $N^2 + M$ constraint equations—one for each commodity, and one for each link. For 10^4 nodes and 10^4 links, the problem has on the order of 10^8 variables and 10^8 constraints. Contemporary solvers often take hours to solve such problems, although the exact runtime depends on problem properties such as sparsity [136].

We can achieve much faster allocation computation times by decomposing the problem; for example, the problem of allocating 10^3 commodities on 10^3 links ($100\times$ fewer variables) is much more tractable. This procedure of breaking up the larger problem into sub-problems *reduces the search space* explored by the solver, since interactions between

all combinations of commodities are no longer considered. Instead, only combinations of *subsets* of commodities are considered, which reduces runtime but also can reduce the quality of the allocation. In light of this, the interaction between commodities needs to be considered carefully to take into account the many global constraints in the original problem, as well as the objective. Additionally, the link capacities need to be distributed appropriately, so that a sub-problem has enough capacity to service its assigned commodities. We find that on large traffic matrices, splitting commodities *randomly* and dividing link capacities equally among the sub-problems reduces the search space of feasible solutions that needs to be considered by solvers, while still ensuring that *some* high-quality feasible points are in the explored search space. This is the main intuition that allows POP to be effective, returning flow allocations of similar quality as the original formulation but faster.

4.2.2 Procedure for POP

The first step of POP is to **`partition`** the traffic engineering problem into smaller sub-problems. The type of partitioning allowed is dependent on the objective and constraints of the allocation problem, and has implications on the runtime speedups and quality of the returned allocation. We can then re-use the map-reduce API [48, 145] (or divide-and-conquer): each of these sub-problems can be solved in parallel (**`map`** step) using the same exact LP as the original problem, and then allocations from the sub-problems can be reconciled into a larger allocation for the entire problem (**`reduce`** step) through a simple summation. We show pseudocode for this in Algorithm 1.

The partitioning step affects the runtime, the reconciliation complexity, and ultimately the quality of the final allocation. For traffic engineering, we use a straightforward approach and divide the commodities randomly into the sub-problems. We find that this partitioning scheme is effective even when the traffic matrix does not exhibit any skew (e.g., the commodities' demands are relatively uniform). With random partitioning, the **`reduce`** step is cheap, which was not the case in NCFlow: there, a more complex reconciliation step was involved to combine the flow allocations from the various sub-problems.

Algorithm 1 POP Procedure.

Input: Commodities $\mathcal{D} = [x_1, x_2, \ldots, x_K]$, links $\mathcal{E} = [e_1, e_2, \ldots, e_M]$, number of sub-problems l, (optional) ratio of extra virtual commodities allowed t.
Return: Allocation for all K commodities, $\mathbf{f}$.

// Optional: make the problem granular if it is not already.
$\mathcal{D}' = \text{SPLIT\_COMMODITIES}(\mathcal{D}, t)$

// This is the **`partition`** *step.*
$[\mathcal{D}'_1, \mathcal{D}'_2, \ldots, \mathcal{D}'_l], [\mathcal{E}'_1, \mathcal{E}'_2, \ldots, \mathcal{E}'_l] = \texttt{partition}(\mathcal{D}', \mathcal{E}, l)$
// For each edge $e \in \mathcal{E}'_j$, the edge's capacity is now c_e/l.

// This is the **`map`** *step, can be performed in parallel.*
for i in `range`(l) **do**
  $\mathbf{f}_i = \text{MaxFlow}(\mathcal{D}'_i, \mathcal{E}'_i)$
  // This could be a different objective, such as MaxConcurrentFlow.
end for

// This is the **`reduce`** *step; flow allocations $\mathbf{f}_i$ are summed up.*
$\mathbf{f} = \sum_i \mathbf{f}_i$

4.2.3 Transformations to Granularize TE Problems

In some cases, it might not be possible to either return a flow allocation that is feasible or high quality by merely assigning each commodity to each sub-problem at random when using the POP procedure. This is notably the case when we have a skewed traffic matrix: when a small number of commodities have large demands that dominate the rest [12]. This is common for certain traffic patterns, such as the Poisson traffic model that we saw in §3.5. If several of these "heavy" commodities were assigned to the same sub-problem, this would immediately lead to sub-optimal total flow; the link capacities assigned to that some problem will not be sufficient. To transform these into granular problems, we propose an algorithm called *commodity splitting* to split these commodities across several sub-problems.

For simplicity when defining the algorithm, we also define the concept of a *splitting attribute* that we will use to determine how to create these virtual commodities and distribute them across several sub-problems. When splitting, it is important that the commodity's

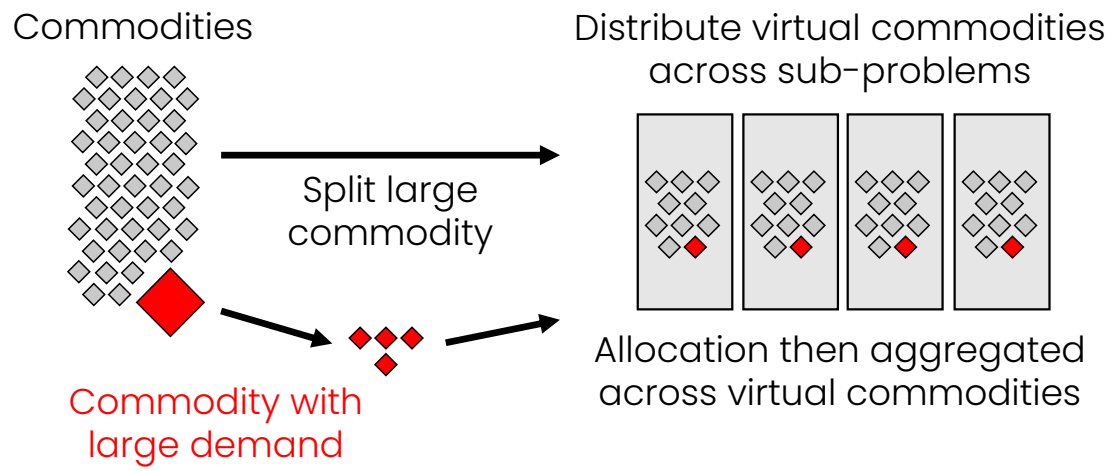


Figure 4.2: Commodity splitting, where we granularize a traffic matrix with skewed demands by splitting the largest commodities until a pre-determined threshold t is met.

other attributes—its source, its target, and its paths—are copied over without change. In our setting, this splitting attribute is the commodity's traffic demand. We then construct a priority queue (heap) of the corresponding attribute values for all clients. Given a threshold t (t is typically a number less than 1) on the maximum number of extra *virtual* commodities allowed, we pop and split commodities off the queue, and then push the new commodities back into the queue. Each split reduces the value of the demand of the popped commodity by a factor of 2. Importantly, each split maintains the feasibility invariant: the coalesced allocation across virtual commodities will still be feasible (since the total sum of the split demands remains the same). By reducing the value of the demand, commodity splitting breaks down large commodities into a collection of smaller commodities with equivalent total demand. The runtime of this algorithm is $O(K \log K)$, where K is the number of commodities, which is cheap compared to the runtime of computing a flow allocation in each sub-problem. Algorithm 2 shows pseudocode, and the procedure is illustrated in Figure 4.2. Empirically, we found that most TE problems are granular enough for POP to work well with 0 split commodities. Moreover, commodity splitting does not adversely impact allocation quality, but it can increase runtime. The hardest problems in our experiments required $t = 0.75$. The optimal value of t is problem-specific, and it is possible that users may have to dynamically adapt t to get the best performance from POP. However, in all

Algorithm 2 Commodity Splitting Algorithm.

Input: Inputs $D = [x_1, x_2, \ldots, x_K]$, ratio of extra virtual commodities t allowed.
Return: Mapping from real to virtual clients $\{x_i \rightarrow [x'_j]\}$.
Initialize queue $\leftarrow$ MAX_HEAP(), mapping $\leftarrow \{\}$.
// Enqueue every commodity based on its demand
For all $i \in \{1, 2, \ldots, K\}$, queue.PUSH($x_i$.demand, x_i).

while len(queue) $\leq (1+t) \cdot n$ **do**
  x_{max} = queue.POP()
  Split x_{max} by demand into two copies x^1_{max} and x^2_{max} (x^1_{max}.demand, x^2_{max}.demand $= x_{\text{max}}$.demand$/2$).

  UPDATE_MAPPING($x_{\text{max}}, [x^1_{\text{max}}, x^2_{\text{max}}]$)
  queue.PUSH(x^1_{max}.demand, x^1_{max}), queue.PUSH(x^2_{max}.demand, x^2_{max})
end while

of the considered production use-cases in our experiments, we found that small values of t that worked well for historical problem instances continue to work well on future problem instances.

The resulting allocation problem after these transformation steps can be granular; if so, we can use POP to solve it. After the `partition` step, we obtain allocations for each virtual variable in the problem. Allocations assigned to virtual variables corresponding to a single commodity need to be summed to obtain the final allocation. We show how this can be incorporated into the full POP procedure in Algorithm 1.

4.2.4 Advantages of POP

Our simple approach provides us with several advantages. To begin with, because POP reuses the underlying problem formulation, it can be applied to a broad class of TE objectives—not only maximum total flow, but also maximum concurrent flow, for example:

$$\begin{aligned} \mathsf{MaxConcurrentFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq & \max \alpha \\ \text{s.t.} \quad & d_k \alpha \leq f_k, \forall k \in \mathcal{D} \\ & \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \end{aligned} \tag{4.1}$$

Similarly, we can apply POP to min-max link utilization:

$$\begin{aligned} \mathsf{MinMaxLinkUtil}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq & \min z \\ \text{s.t.} \quad & \sum_{\forall k, p \in \mathcal{P}_k, e \in p} f_k^p \leq z, \forall e \in \mathcal{E} \\ & \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \end{aligned} \tag{4.2}$$

Here, we see that POP's flexibility is analogous to the linear program itself: the constraints and the underlying problem definition do not have to change—only the objective function changes. By contrast, these objectives have traditionally required different approximation algorithms [53, 80].

Another advantage of POP is its composability: it can be used in an outer loop as a simplifying step for a downstream heuristic or approximation algorithm. For example, one might use POP to initially partition the TE problem, and then use Fleischer's algorithm to solve the individual sub-problems. POP could also be combined with different path selection algorithms. Lastly, like NCFlow, POP exposes a tunable knob—the number of sub-problems l—that lets the user explicitly trade-off between flow allocation quality and runtime.

4.2.5 When Does POP not Apply to TE?

Note that POP is not applicable for every variant of the TE problem. For example, suppose our problem included hard constraints like "flows A and B should/should not use the same link." This would not align well with randomly partitioning the commodities (e.g., random partitioning could drop flows A and B into different sub-problems when flows A and B

Topology	# Nodes	# Edges
Kdl	754	1790
Cogentco	197	486
UsCarrier	158	378
Colt	153	354
GtsCe	149	386
TataNld	145	372
DialtelecomCz	138	302
Deltacom	113	322

Table 4.1: The WAN topologies used to benchmark POP. Like Table 3.3, the networks in blue were obtained from the Internet Topology Zoo [82] at http://www.topology-zoo.org/.

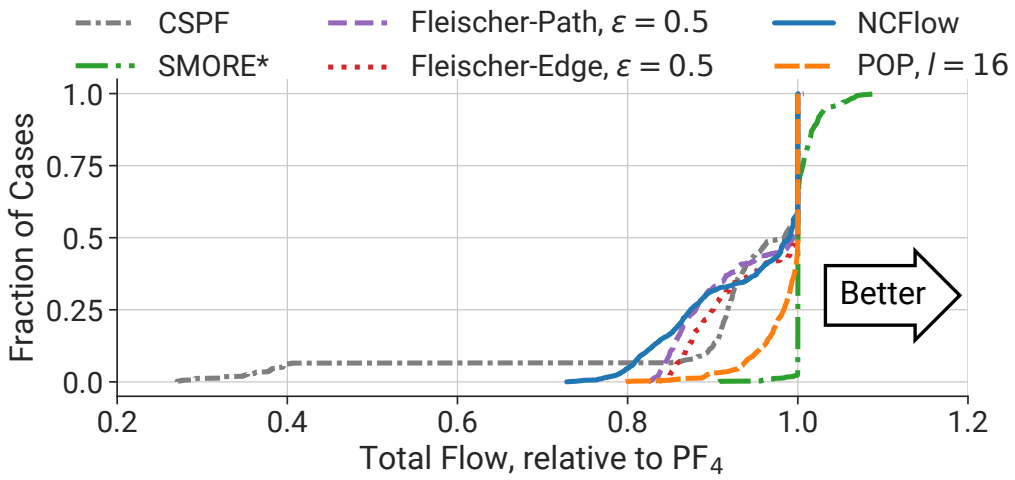
need to use the same link); smarter partitioning algorithms can mitigate this by considering affinity between flows, but supporting these is left to future work.

4.3 Evaluation

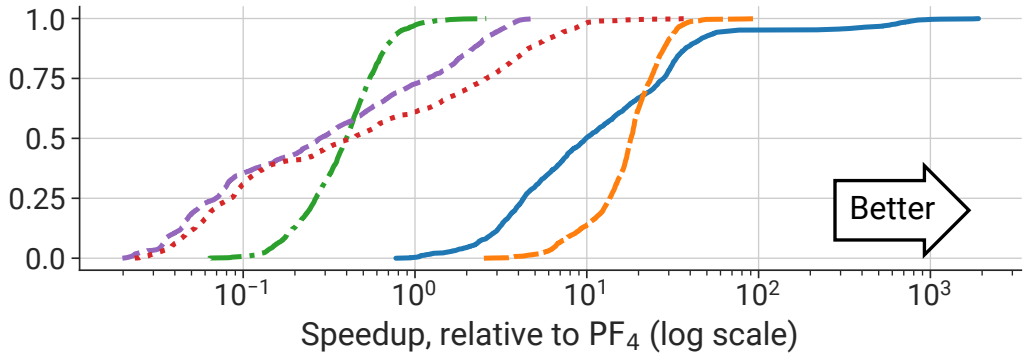
In this section, we evaluate POP against NCFlow and the state of the art in TE. We aim to answer the following questions:

1. What is the effect of POP on flow allocation quality and execution time for traffic engineering? How does it compare to relevant heuristics?
2. Does POP work across a range of objective functions?
3. How effective are POP's commodity splitting optimization in generating high-quality flows?
4. How does random partitioning compare to other more sophisticated problem partitioning strategies?

We evaluate POP using the same set of traffic matrices as NCFlow, but on a subset of the topologies from Table 3.3. (See Table 4.1.) Unlike NCFlow, our results span three different TE objectives: maximize total flow, maximize concurrent flow, and min-max link utilization.



(a) CDF of total flow relative to PF_4



(b) CDF of speedup relative to PF_4

Figure 4.3: CDFs comparing POP vs NCFlow with state-of-the-art methods. Like NCFlow, POP offers a substantial runtime speedup, with only a modest decrease in total flow.

We first present end-to-end experiments, then present some micro-benchmarks that examine the impact of various algorithmic contributions in POP.

4.3.1 Comparing POP to the State of the Art

We first demonstrate POP's end-to-end effectiveness. In our experiments, the total number of threads given to solvers for our baselines and POP are the *same*. If l sub-problems are solved in parallel when using POP, each sub-problem uses $1/l$ of the number of threads. Unless otherwise noted, we benchmark POP with $l = 16$ sub-problems.

We tested POP on the topologies shown in Table 4.1 from the Topology Zoo repository [82]. For each topology, we benchmarked POP on the same set of synthetic traffic matrices previously seen in §3.5. As previously noted, these traffic matrices were generated using several traffic models: Gravity [18, 123], Uniform, Bimodal [18], and Poisson. Because Poisson represents a skewed workload, where a small percentage of commodities dominate the network demand. For this workload, we use the commodity splitting algorithm from §4.2.3 to improve flow quality. We do not use commodity splitting for the other traffic matrices.

Maximum Total Flow. Figures 4.3a and 4.3b show cumulative density functions (CDFs) of the relative total flow and speedup ratio for POP and NCFlow and several baselines. These two figures are extensions of Figure 3.10, with the exception benchmark on the 8 topologies in Table 4.1.

We see that POP, the densely dashed orange lines in both figures, almost always allocates at least 90% of PF_4's total flow, while achieving large speedups. In the median case, POP achieves 99.9% of the flow and is over $18\times$ faster than PF_4. By contrast, NCFlow achieves 99.0% of the flow, with a median speedup of $9.8\times$ on the same cohort of traffic matrices. However, NCFlow seems to outperform POP at the tail of the speedup CDF: its maximum speedup is $1906\times$, while POP's is $98\times$.

Figure 4.4 shows the trade-off between runtime and allocated flow on the Kentucky Data Link network (Kdl in Table 4.1), which has 754 nodes and 1790 edges spanning the Eastern half of continental USA. We instantiated over 5×10^5 demands, with 4 paths per commodity in the network. The flow allocated by POP is within 1.5% of optimal when using 64 sub-problems, yet $100\times$ faster than the original problem. Again, POP also compares favorably to CSPF [58] and NCFlow.

Figure 4.5 shows the improvement in allocation quality and runtime compared to the original LP formulation presented in §6.4 with POP using 16 sub-problems. Each point in the scatterplot represents a different topology and traffic matrix. We see larger speedups for the larger Kdl topology. We used commodity splitting with a threshold (t) of 0.75 for the Poisson traffic matrices (where some commodities have large demands), and no commodity splitting for the other traffic models, which were granular out of the box.

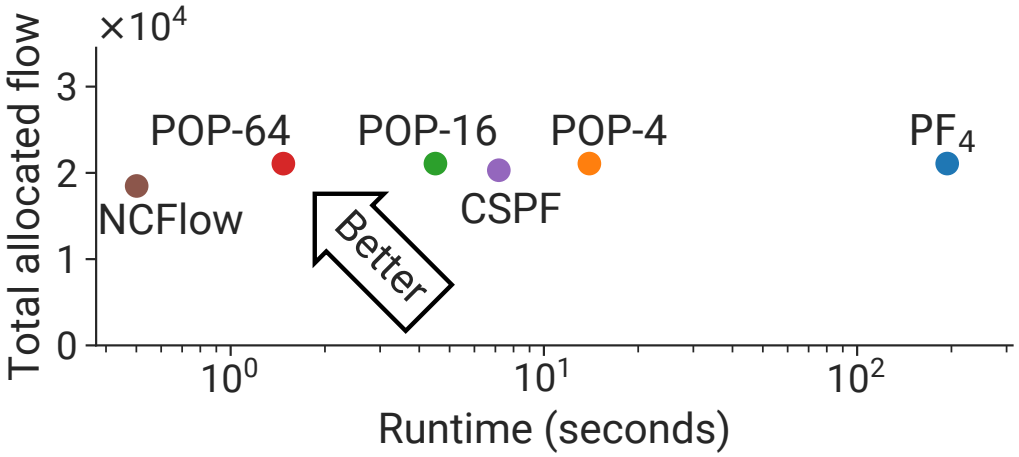


Figure 4.4: Results for the Maximum Total Flow problem for traffic engineering for a single topology and traffic matrix. The scatterplot shows runtimes and total allocated flow for the formulation shown in Equation 2.2 (PF_4) and its POP variants, as well as CSPF and NCFlow.

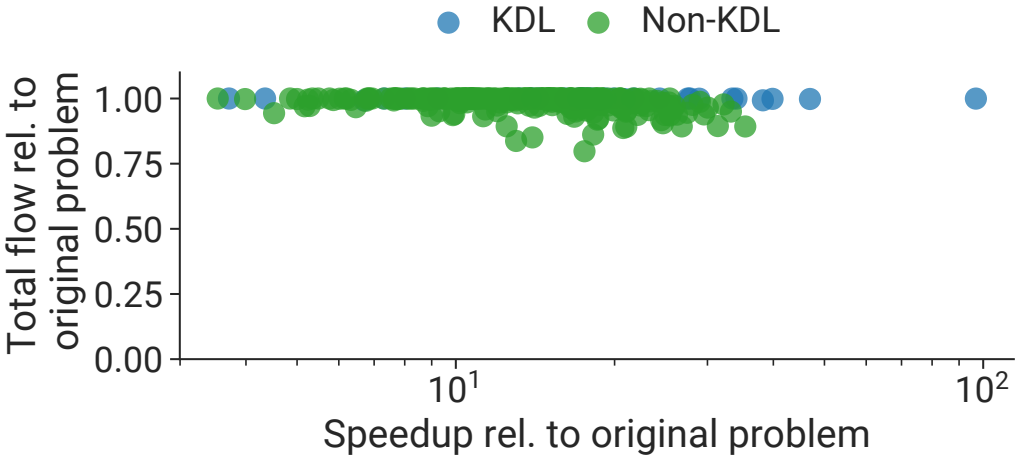


Figure 4.5: Results for the Maximum Total Flow problem for traffic engineering for multiple topologies and traffic matrices. The scatterplot shows runtimes and allocated total flow for POP-16 across 275 experiments, separated by large (Kdl) and small (non-Kdl) topologies.

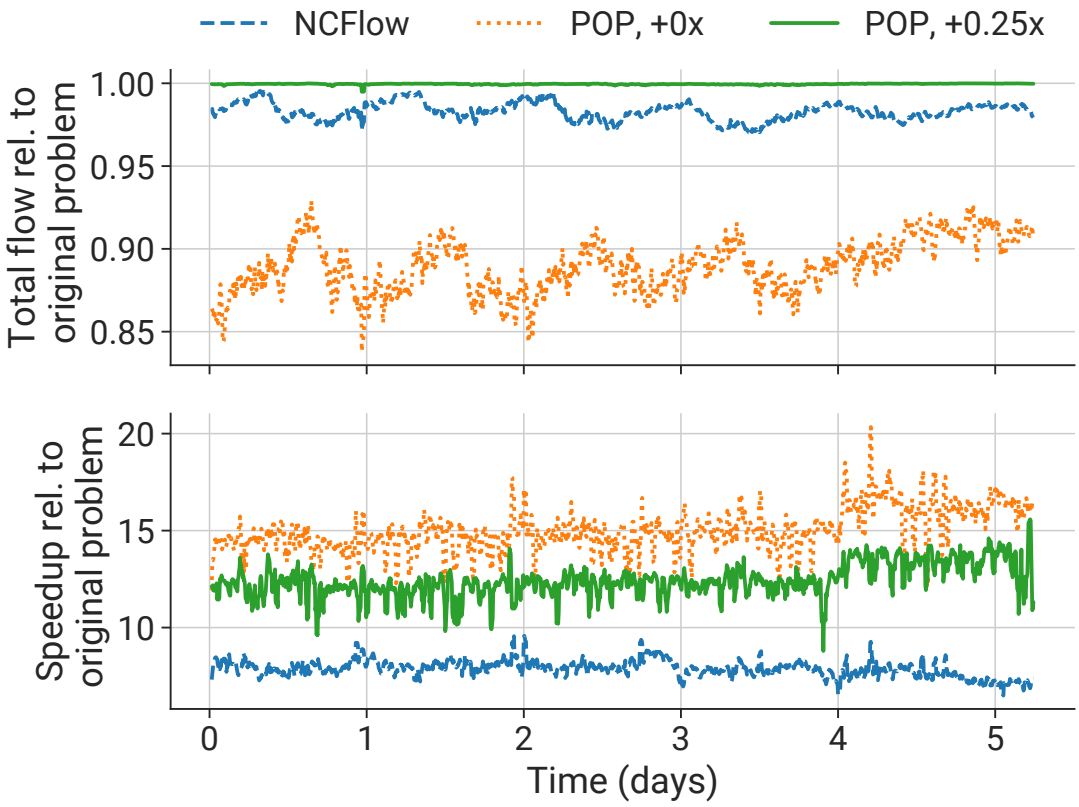


Figure 4.6: Allocated flow and speedup relative to original problem on a 5-day sequence of real-world traffic matrices from a private WAN with 100s of nodes and edges. With commodity splitting ($t = 0.25$), POP allocates $>$99% of the total flow with a 12.5$\times$ median speedup.

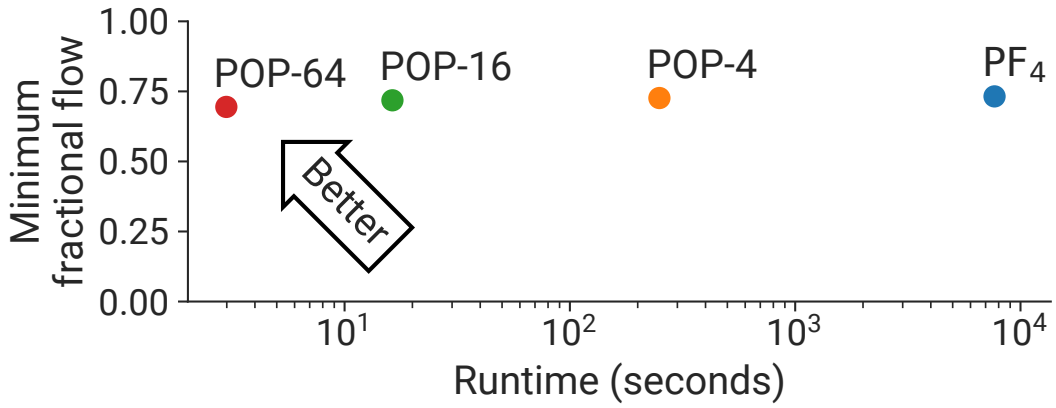


Figure 4.7: Results for the Maximum Concurrent Flow problem for traffic engineering for a single topology and traffic matrix. The scatterplot shows runtimes and minimum fractional flow for the formulation shown in Equation 4.1 (PF_4) and its POP variants.

We also ran experiments on the same set of real-world traffic matrices we saw in Figure 3.13. Figure 4.6 plots the moving average (over 5 windows) of the total flow and speedup relative to the original problem for NCFlow, POP with no commodity splitting, and POP with $t = 0.25$ commodity splitting. Without commodity splitting, POP achieves significant speedups ($15\times$ in the median case) compared to the original problem, but allocates 89.1% of the total flow in the median case. However, POP with commodity splitting nearly matches the total flow allocated in the original problem (99.9% in the median case), while still achieving a median $12.5\times$ speedup.

Maximum Concurrent Flow. Similarly, we benchmarked POP on the Maximum Concurrent Flow objective using the same set of topologies and traffic matrices. Figure 4.7 shows the trade-off between runtime and minimum fractional flow on the Kdl topology, using the same traffic matrix in Figure 4.4. The objective value realized by POP is again within 1.5% of optimal when using 64 sub-problems, yet $1000\times$ faster than the original problem. As before, we use commodity splitting with a threshold of 75% for the Poisson traffic matrices, and no commodity splitting for the other traffic matrices.

Min-Max Link Utilization. Lastly, we benchmarked POP on the Min-Max Link Utilization objective using the same set of topologies and traffic matrices. Figure 4.8 shows the

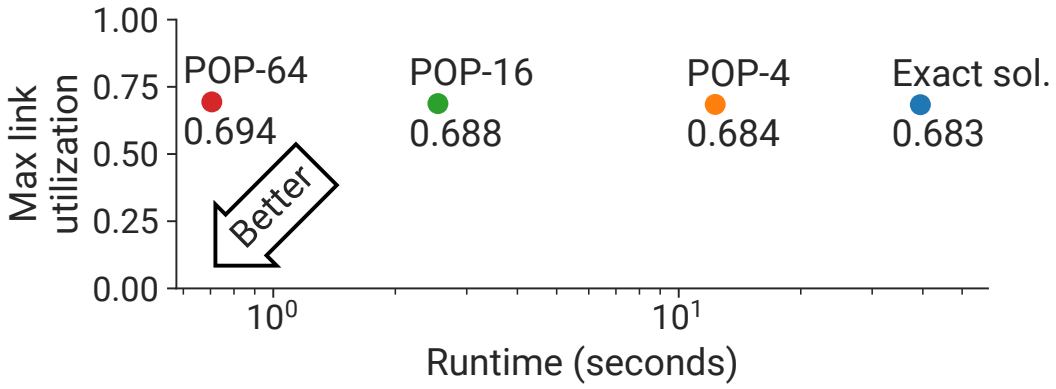


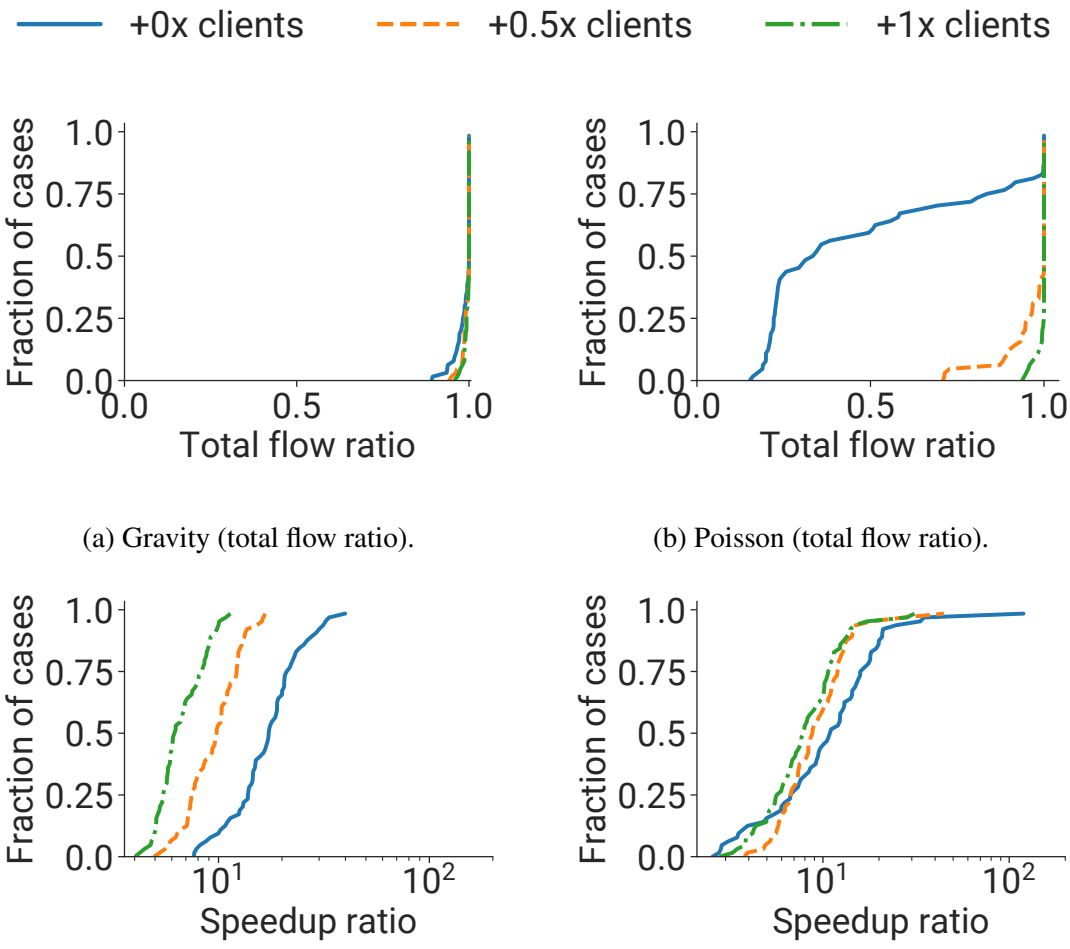
Figure 4.8: Results for the Min-Max Link Utilization problem for a single topology and traffic matrix. The scatterplot shows runtimes and maximum link utilization for the formulation shown in Equation 4.2 (PF_4) and its POP variants.

trade-off between runtime and maximum link utilization again on the Kdl topology, using the same traffic matrix in Figure 4.4. The objective value realized by POP is again within 1.5% of optimal when using 64 sub-problems, yet $56\times$ faster than the original problem. As before, we use commodity splitting with a threshold of 75% for the Poisson traffic matrices, and no commodity splitting for the other traffic matrices.

4.3.2 Effectiveness of Commodity Splitting

Figure 4.9 shows the effect of commodity splitting on total flow and runtime when using POP with 16 sub-problems, on a traffic engineering problem with "large" commodities (Poisson traffic model) as well as a more typical set of commodities (Gravity traffic model) and a Maximum Total Flow objective. The figure shows separate cumulative distributions of approximately 100 different experiments for each traffic model and commodity splitting threshold (t in Algorithm 3).

We see that with skewed traffic (Poisson traffic model) and no commodity splitting, the total flow is typically far from optimal. Commodity splitting drastically increases the median relative total flow from 0.2 to near 1.0 for these problems, at the cost of some runtime overhead (due to an increase in the number of variables). In contrast, the problems with Gravity traffic get near-optimal allocated flow without commodity splitting.



(a) Gravity (total flow ratio).

(b) Poisson (total flow ratio).

(c) Gravity (speedup ratio).

(d) Poisson (speedup ratio).

Figure 4.9: Comparison of POP-16 relative to original problem for the Maximum Total Flow objective, across different levels of additional split commodities ($0\times$ to $1\times$) and traffic matrices from two traffic models: Gravity and Poisson (which is skewed).

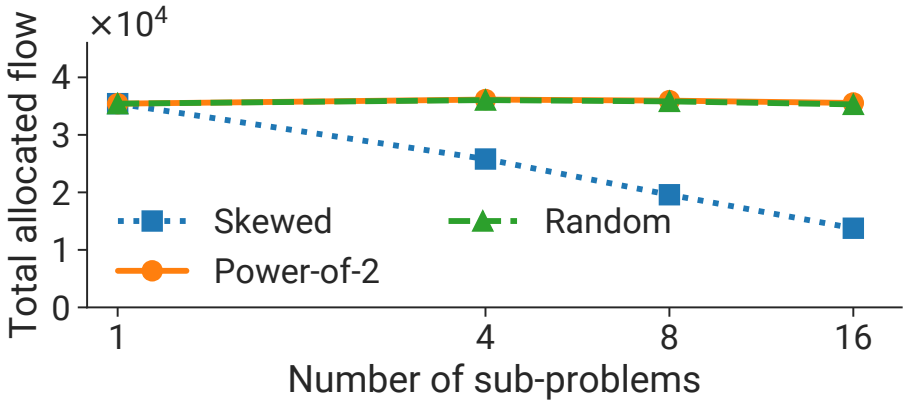


Figure 4.10: Performance comparison of various partitioning algorithms for the Maximum Total Flow objective in traffic engineering. Power-of-2 partitioning is nearly identical to random partitioning.

4.3.3 Alternatives to Random Partitioning

We implemented several algorithms to partition commodities into sub-problems, and compared them to random partitioning. Among these is a power-of-2 partitioning algorithm that tries to assign each commodity sequentially to one of two randomly chosen sub-problems using "distributional similarity" to the original problem as the metric. We also implement a skewed partitioning algorithm that deliberately creates skew among sub-problems to show the impact of bad partitions. Figure 4.10 shows the impact of these partitioning algorithms on the quality of allocation returned by POP on a traffic engineering problem. We see that random partitioning performs about as well as the more sophisticated power-of-2 partitioning, while skewed partitions have poor performance (skewed causes link congestion around certain nodes in the WAN).

4.4 Related Work and Discussion

In this section, we discuss other techniques for traffic engineering that share lineage with POP.

Randomized Algorithms in Networking. Randomized approaches have seen success in other important networking tasks as well. For example, in data center networking [129], random graph topologies work surprisingly well compared to commonly-used structured topologies such as FAT-trees. In load-balancing algorithms [99], assigning jobs to the least-loaded of just two randomly selected servers in a cluster can drastically reduce the probability of overloading a server.

Additionally, Valiant Load-Balancing (VLB) [146] uses a simple randomized approach for routing traffic: the source node samples an intermediate node to forward traffic to (e.g., using CSPF or ECMP), and the intermediate node then forwards the traffic to the correct destination. VLB has been helpful in mitigating link failures, particularly in data center networks [66]. However, it is not used heavily for traffic engineering in large-scale WANs today.

So called "oblivious" or semi-oblivious routing schemes have become popular in the traffic engineering community in the last decade, which also leverage randomization. Algorithms such as SMORE [86] use Randomized Routing Trees (RRTs), a technique developed by Räcke [118]. These RRTs can be used to improve path selection for commodities, since they produce paths that are low-stretch, diverse, and naturally load-balanced. This differs from POP, which is agnostic to the path selection algorithm.

Approximation Algorithms for TE. FPTAS algorithms [9] return results with a guaranteed approximation ratio and run in polynomial time over this approximation factor. Proving an approximation ratio with POP is hard since we apply POP to many different problems with various structures, as opposed to designing a problem-specific approximation algorithm. Many such algorithms exist for various traffic engineering objectives [25, 53, 59, 80]. We benchmark against one of these algorithms—Fleischer's [53]—in our evaluation and show that POP achieves a better trade-off between optimality and efficiency.

4.5 Summary

In this section, we showed how commodity-based partitioning can be exploited to efficiently solve traffic engineering problems. Our technique, POP, achieves strong results

by randomly partitioning the global TE problem into smaller, independent sub-problems, which reduces the problem's complexity and allows us to find a feasible flow allocation in an embarrassingly parallel fashion. Furthermore, POP retains the global problem's optimization objective and constraints, which makes it reusable across multiple TE objectives. We found that POP achieved runtime improvements of $18\times$ in the median case with small optimality gap, and it also outperformed greedy ad-hoc heuristics and other approximation algorithms. We hope this work motivates others to consider POP for their large-scale traffic engineering needs.

Chapter 5

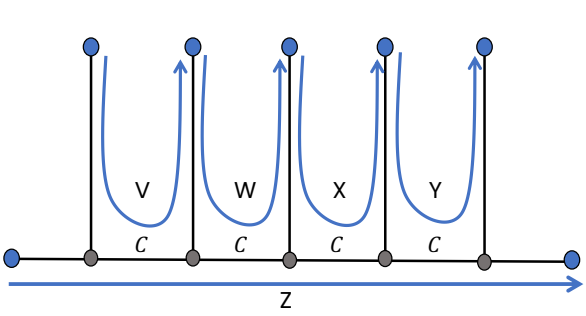
NCFlow vs POP

In this chapter, we compare NCFlow and POP head to head and enumerate their relative strengths and weaknesses. So far, we have discussed each of these algorithms in isolation, and we have also seen experimental results that directly compare POP and NCFlow. However, we have not yet elucidated *how* these results were achieved by both algorithms. More specifically, we have not yet answered the question: when does NCFlow outperform POP—on what types of topologies and/or traffic matrices? Similarly, when does POP outperform NCFlow? By answering these questions, we can deepen our understanding of these algorithms and develop an appreciation for when to use one versus the other.

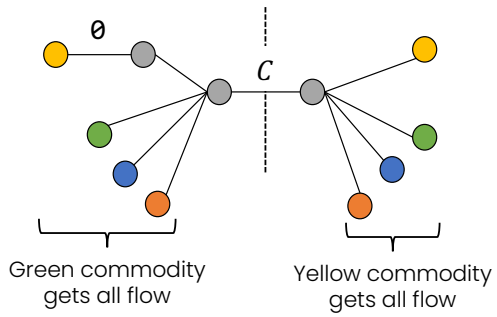
First, we examine two small toy examples—one that demonstrates POP's shortcomings, and one that demonstrates NCFlow's. We provide these examples to help the reader developer better intuition for the matter at hand. Then, we build off of these examples and discuss some experimental results that clarify and solidify the key characteristics that determine NCFlow's and POP's relative performances. Finally, we summarize the differences between the two algorithms more broadly in our concluding section.

5.1 When Does POP Underperform?

To better understand POP's weaknesses, we construct a simple topology and traffic matrix in Figure 5.1a. A detailed walkthrough of this scenario will illustrate how POP can behave sub-optimally for certain inputs.



(a) Failure case for POP. The optimal maximum total flow is $4C$, which NCFlow will match. POP, however, will allocate $2.03C$ in expectation.



(b) Failure case for NCFlow. The optimal maximum total flow is C, and POP will allocate $0.93C$ in expectation. In the worst case, NCFlow will allocate 0 flow.

Figure 5.1: Failure scenarios for POP and NCFlow. For each scenario, assume all commodities have the same demand $d \geq C$.

In this example, commodities V, W, X, Y, and Z all have the same demand $d \geq C$, where C is the capacity on each link. By inspection, we can conclude that, for the maximum total flow objective, the optimal allocation is $4C$: V, W, X, and Y should allocate C flow, and Z should get 0.

If we apply NCFlow on this problem, with a simple partitioning of two clusters that separate commodities V and W into one cluster, commodity Y in another, and commodities X and Z the remaining inter-cluster commodities, then we can also see through a simple simulation of NCFlow's algorithm that it will also arrive at $4C$, matching the optimal allocation.

Now how does POP perform, with $l = 2$ sub-problems[1] on this toy example? Without loss of generality, let's assume that commodities V and W are assigned to the first sub-problem, and commodities X, Y, and Z are assigned to the second sub-problem. Both sub-problems now have capacities $C/2$ across all links. In the first sub-problem, commodities V and W will saturate their links, giving us a total flow of $C/2 + C/2 = C$. Similarly in the second sub-problem, commodities X and Y saturate their links, also achieving a total flow of C. (Z is given 0 flow.) The total flow, then, across all sub-problems is $2C$, half of

[1] Assume that no commodity splitting is applied.

the optimal allocation.[2]

When we divide the link capacities across the two sub-problems, we restrict the capacity budget for the sub-problem's assigned commodities—we are effectively forcing the first sub-problem's commodities to share the links with the second sub-problem's. But, in this situation, the optimal flow is achieved when one commodity takes the *entire* link, as we outlined earlier.

5.2 When Does NCFlow Underperform?

Let us now examine NCFlow to better comprehend its weaknesses. Similar to our analysis for POP, we devise a small topology and traffic matrix in Figure 5.1b. Once again, we walk through this example step by step to see how NCFlow can arrive sub-optimal allocations.

In this small "dumbbell" topology, there are four commodities—yellow, green, blue, and orange—that must all pass through a single link with capacity C. As was the case in the previous section, all four commodities have demand $d \geq C$. However, the yellow commodity must first pass through an additional link, which has capacity 0. (All other unmarked links have infinite capacity.) Effectively, the yellow commodity must get 0 flow no matter what.

Because of the bottleneck link with capacity C, the optimal flow allocation is clearly C as well: any commodity (other than yellow) can take the link's capacity, or it can be shared evenly between the green, blue, and orange. Because the link can be shared, this bodes well for POP. In fact, if we apply POP with $l = 2$ on this problem, we can see that it should match the optimal result C with high probability.[3]

How will NCFlow perform on this problem? Suppose we cluster the topology based on the dashed line drawn in Figure 5.1b: one cluster on the left, and another on the right. The left cluster will, of course, not allocate any flow to the yellow commodity in Max-ClusterFlow. However, the right cluster certainly could; if that were to occur, then, after

[2]Other possible partitionings will also yield $2C$, except for one: $\{V, W, X, Y\}$ and $\{Z\}$, which would allocate $2.5C$. This partitioning can occur with probability $1/15$, which means that, in expectation, POP will allocate $2.03C$.

[3]The only exception is if the random partitioning turns out to be $\{\text{green}, \text{blue}, \text{orange}\}$ and $\{\text{yellow}\}$; in that situation, the second sub-problem would allocate 0 flow, and the total flow would be $C/2$. So, in expectation, POP will allocate $0.93C$.

SrcTargetMax is executed, NCFlow would end up allocating 0 flow.

Note that this does not always happen: it is also certainly possible that, under NCFlow, the two clusters end up agree on which commodity (or commodities) get assigned the flows. But, without some canonical ordering (which is non-trivial to design), this problem can arise with high probability. Because of the *intra*-cluster bottleneck on an *inter*-cluster demand, NCFlow now has the potential to create a disagreement that would never occur in the original formulation of the problem.

To conclude, we summarize the key difference between NCFlow and POP: for certain topologies and traffic matrices, the optimal approach is to let a subset of the commodities take all the capacity. In those situations, NCFlow will likely outperform POP. However, those circumstances do not always hold true; in fact, the exact opposite may be in effect: to achieve optimality, all of the commodities must share the link capacities amongst one another. Under those conditions, we expect POP to outperform NCFlow.

5.3 Experimental Results

We have illustrated how POP and NCFlow can both under-allocate flows in certain scenarios. However the two examples we have given are both small topologies and traffic matrices; a skeptic could argue that these are fairly contrived and do not necessarily capture how both of these algorithms will behave at scale.

The skeptic's critique has merit. To remedy this, we conducted an experiment that captures the broader characteristics of each scenario *in situ*—i.e., we recreate these scenarios within the topologies used in our evaluation of NCFlow and POP from section 4.3.

For each topology, we generated two 'imbalanced" traffic matrices: one that is heavily skewed to commodities whose sources and targets are close to one another in the topology, and another that is skewed to commodities who sources and targets are far apart from one another. We do this programmatically using the following procedure:

1. Using the computed path sets for the topology, we calculate the average hop length for each commodity's set of paths and sort the commodities based on this metric.

Topology	Heavy demand on "near" commodities		Heavy demand on "far" communities	
	NCFlow	POP	NCFlow	POP
Kdl	**0.99**	0.59	0.57	**0.90**
Cogentco	**0.99**	0.60	0.67	**0.92**
UsCarrier	**0.99**	0.59	0.75	**0.93**
Colt	**1.0**	0.61	0.80	**0.94**
GtsCe	**0.99**	0.61	0.62	**0.93**
TataNld	**0.99**	0.60	0.80	**0.89**
DialtelecomCz	**1.0**	0.58	**0.88**	0.72
Deltacom	**0.98**	0.62	0.70	**0.93**

Table 5.1: NCFlow vs POP on "imbalanced" traffic matrices for topologies from Table 4.1. We show the relative total flow compared to PF_4 for each technique; the bolded numbers signify which one outperformed the other (higher is better). NCFlow is better than POP when nearly all demand in the traffic matrix is assigned to commodities that are near one another in the topology, while the reverse is true when the demand is concentrated amongst commodities that are far apart from one another.

2. After sorting, we select the top/bottom 50% of commodities and assign them a demand equal to the maximum capacity of any link in the topology. The remaining 50% of commodities are given demand 0, which gives us a complete traffic matrix.

Using this procedure, we can generate two types of traffic matrices: one where where the demands are "far apart" in the topology (the top 50%), and one where the demands are "near" one another (the bottom 50%). In both cases, the topology will be heavily oversubscribed with traffic, since, by definition, each demand could saturate any link in the topology. The difference in each traffic matrix comes down to *where* the saturation will occur. More specifically, our setup should lead to the following outcomes:

- The "near" traffic matrix will ideally capture POP's failure scenario: many bottlenecked single-path flows. By contrast, NCFlow will thrive.
- The "far apart" traffic matrix will capture NCFlow's failure scenario: many inter-cluster demands with many intra-cluster bottlenecks. POP, on the other hand, will

NCFlow	POP
Geographic partitioning	Commodity-based partitioning
Replaces original LP with a new set of LPs (see Figure 3.4)	Reuses original LP, but discards variables via partitioning
Successful when the optimal approach requires a small subset of the commodities to take all the capacity	Successful when the optimal approach requires a large portion of the commodities to share the link capacities (max total flow only)
Excels when demand is concentrated on "near" commodities	Excels when demand is concentrated on "far apart" commodities
Paths computed based on clustering	Path-agnostic, can use any set of paths
Reduces # of FIB entries	No impact to FIB entries
Maximum Total Flow objective only	Supports multiple objectives

Table 5.2: Summary of the differences between NCFlow and POP.

perform well under these circumstances.

The results of this experiment are shown in Table 5.1, where we show the relative total flow of each technique compared to PF_4. As we can see, NCFlow significantly outperforms POP on the "near" traffic matrix, which matches our intuition. When the commodities with the largest demands are close to one another in the topology, there are fewer paths to choose from. Thus, the likelihood of a single path—or even a single link—becoming a shared bottleneck amongst many commodities increases, which, as we saw in Figure 5.1a, leads to poor performance with POP.

5.4 Summary

In this chapter, we analyzed the strengths and weaknesses of NCFlow and POP and illustrated how they both behave under certain traffic conditions. We showed that NCFlow excels when the optimal approach requires a small subset of the commodities to take all

the capacity, and this is much more likely to occur when the demand is concentrated on "near" commodities. This directly contrasts with POP, which excels when the optimal approach requires a large portion of the commodities to share the link capacities. Those circumstances typically arise when the demand is heavily concentrated on "far apart" commodities in the topology.

There are more differences between the two techniques, which we did not fully cover in this section but were previously discussed elsewhere. We summarize those differences in Table 5.2. Ultimately, neither technique strictly dominates the other, and we hope that our discussion clarifies and distinguishes the nuances between the two.

Chapter 6

Additional Applications of POP

6.1 Introduction

In this chapter, we discuss how to extend POP beyond traffic engineering to a broader set of resource allocation problems in computer systems. In the course of our work, we discovered that we could apply these techniques to similar problems in other domains. Thus, we reframe our commodity-based partitioning approach as a general-purpose strategy that can be applied to generic clients and resources in a resource allocation problem.

As computer systems become larger and workloads on these systems continue to grow, it has become common for systems to be shared among multiple users. As a result, deciding how resources (e.g., GPUs, links, servers) should be shared amongst various clients while optimizing for many macro-objectives is important across a number of domains (e.g., cluster scheduling and load balancing, in addition to traffic engineering).

Like TE, these resource allocation problems can often be formulated as mathematical optimization programs [12, 65, 85, 93, 103, 106, 117, 120, 131]; the output of these programs is the allocation of resources (e.g., accelerators, servers, or network links) to each client (e.g., jobs, data shards, or traffic commodities). Unfortunately, solving these mathematical programs can be computationally expensive. The worst-case complexity for linear programs is approximately $O(n^{2.373})$ [44, 91], where n is the number of problem variables (even though LPs can often be solved faster depending on problem structure), and integer-linear programs are even more expensive. Mathematical programs for resource allocation

can have millions of variables (e.g., one variable for every `(client,resource)` pair) for large-scale systems, leading to long solution times depending on the numerical solver used (e.g., 8 minutes for a cluster with 1000 jobs using SCS [108, 109]). Moreover, allocations often need to be recomputed frequently to keep up with dynamic changes in the system, which we've also seen in the TE problem setup. Consequently, production systems such as the Accordion load balancer [127] for distributed databases and the Gavel job scheduler [103] hit performance bottlenecks when the numbers of clients and resources increase.

Thus, the conventional wisdom in the systems community is that solving these programs *directly* often takes too long. Instead, production systems and researchers frequently use heuristics that are cheaper to compute. It is common to see some version of the following statement in a research publication:

> "Since these algorithms take a long time, they are not practical for real-world deployments. Instead, they provide a baseline with which to compare faster approximation algorithms." – Taft et al. [131].

The partition-placement algorithm in E-Store [131], the space-sharing-aware policy in Gandiva [142], and cluster management policies to allocate resources to containers in systems like Kubernetes [4], DRS [69], and OpenShift [8] all rely on heuristics. However, prior work shows that these heuristics are hard to maintain as problems scale and inputs change [130], are far from optimal (Figures 6.1 and 6.8), and often do not extend to slightly modified objectives.

Although it seems that large optimization problems are too expensive to solve directly, we observe that many allocation problems in computer systems share several exploitable properties that we also observed in the TE setting: the number of clients and resources is large, each client requests a small fraction of the total number of resources, and resources are fungible or substitutable (i.e., a job can make similar progress using *different* resources). We propose that POP can be applied to any such granular allocation problem. As was the case with traffic engineering, POP can quickly compute allocations by reusing the original optimization problem formulation on subsets of the input. On several granular optimization problems outside of TE, we found that POP can give close-to-optimal results with orders-of-magnitude faster runtimes than the full formulations. Importantly, since POP reuses the

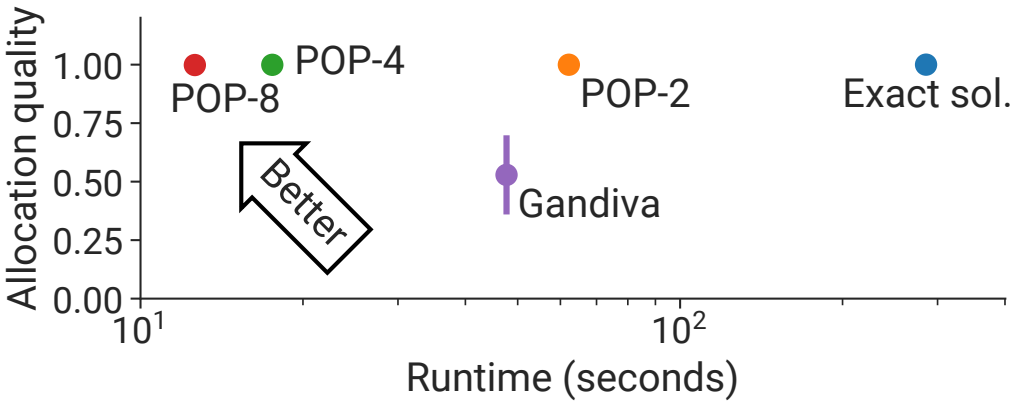


Figure 6.1: Comparison of Gavel's fair-sharing policy compared to its POP variants and Gandiva [142] on a GPU cluster. The scatterplot shows runtimes and mean allocation quality across 2048 jobs on a cluster with 1536 GPUs. POP-l uses l sub-problems.

original problem formulations, it can be implemented in only a few lines of code.

The simplest way to apply POP is to divide clients and resources among l identical copies of the given optimization problem (each with a subset of the clients and resources). Each sub-problem has fewer equations and variables, leading to a super-linear runtime speedup. The sub-problems can also be executed in parallel. The overall allocation is a union of the allocations from the individual sub-problems. Our results show that randomly and evenly dividing clients and resources among sub-problems works well when clients are numerous and individually use only a small fraction of all resources. Empirically, we show that POP's resource allocations are nearly optimal on several optimization problems, including using real-world inputs. We also prove that the probability of a large optimality gap is small given an allocation problem with certain simple properties. POP has structural similarity with the first step of "primal decomposition" in convex optimization [30], but can be applied to a broader set of problems than those amenable to primal decomposition (separable objective, coupled constraints). We note that there can be other ways to POP an optimization problem, but these are beyond the scope of this chapter.

In the wild, allocation problems do not always fit the definition of granular as presented above, e.g., a cluster scheduling problem could have "large" clients (jobs) with substantial hardware demands, or a client might have to use a particular resource (e.g., a specific GPU). Fortunately, in some cases, we can transform the problem into a granular problem using

two granularization techniques: *client splitting*[1] and *resource splitting*. The "large" clients, which individually require a sizable fraction of total resources, can be split into multiple virtual clients who each receive partial allocations from multiple sub-problems. Since the number of "large" clients is small, by definition, POP's sub-problems remain small and still achieve a sizable runtime speedup. Similarly, resources can be split into multiple virtual resources, each with a fraction of the full resource's capacity.

POP cannot be applied to every allocation problem in systems because some problems are not granular or require a non-trivial partitioning into sub-problems (e.g., due to constraints). We discuss examples of such problems in §6.6.

Nevertheless, we found that POP is effective on a wide range of important problems in recent computer systems research. We have already shown its effectiveness on traffic engineering problems for three different objectives; in this chapter, we evaluate POP on four more allocation objectives across two different domains: cluster scheduling and load balancing. POP achieves empirical runtime improvements of up to $100\times$ compared to the original optimization problem formulations while staying within 1.5% of optimal, and even up to $20\times$ faster and $1.9\times$ higher quality than heuristics. We integrated POP into real systems like Gavel, and found that downstream metrics like average job completion time and makespan are unaffected by using POP. We also found granularization useful in using POP to compute high-quality allocations for initially non-granular problems.

6.2 Granular Allocation Problems

Computer systems are often shared among *clients* from multiple users (e.g., jobs in a cluster scheduler, commodities in a wide-area network). These clients might then request *resources* (e.g., GPUs or bandwidth on a network link) from a central resource allocator, which determines how to map resources to clients. Resource allocation problems have three main components:

- **Search Space of Allocations:** Allocations specify how resources should be shared

[1] In this chapter, "client splitting" effectively represents a more general framing of commodity splitting.

between clients. In cluster scheduling, an allocation can specify the fraction of wall-clock time each active job should spend on different types of resources (e.g., types of GPUs like K80, P100, V100, A100). In query load balancing, an allocation can specify which servers should store which shards of data. Allocations can also reason through the interactions between clients on different resources (e.g., the time fractions *pairs* of jobs should spend on various resources [103, 142]).

- **Objectives:** The objective that an optimization problem maximizes or minimizes is a function over the allocation, and specifies the metric (e.g., dollar cost, number of queries) that needs to be optimized in solving the allocation problem. We observe that these functions are typically a max or a sum over functions of per-client allocations. They can, however, be other arbitrary functions as well; convex functions are generally easier to optimize.

- **Constraints:** Most allocation problems also specify constraints to ensure that both clients and resources are not over-allocated (e.g., the total time fraction given to a single job across resource types cannot exceed 1.0) and that various invariants are maintained. These are specified as functions over the allocation A.

The goal of a resource allocation problem is to find the allocation value that is feasible (respects the provided constraints) and optimizes the provided objective.

We can then say an allocation problem is **granular** if:

- **Condition 1:** The number of clients and resources is large (on the order of 100s or more).

- **Condition 2:** Each client requests an insignificant fraction (e.g., $< 1\%$) of the total available resources.

- **Condition 3:** Resources are fungible or substitutable. In other words, if a client c is given resource r as part of an allocation A, there are multiple other resources $r' \neq r$ such that switching c to r' gives an allocation A' with similar objective value ($f(A) \approx f(A')$).

- **Condition 4:** If the resource allocation problem considers interactions between multiple clients (e.g., two jobs on the same server), then client combinations should be fungible or substitutable too.

As we show in §6.4, resource allocation problems in a number of different domains like traffic engineering, cluster scheduling, and load balancing, are granular. Furthermore, in certain cases, problems that violate some of these conditions can be made granular through granularization transformations (client and resource splitting in §6.3.3).

For example, in Gavel [103], a cluster scheduler for machine learning training workloads on clusters of GPUs, each job (client) requests a prescribed number of a resource (e.g., a specific kind of GPU) to make progress. Each job requests a small fraction of the total number of GPUs available in the cluster, and can be run on different types of GPUs with varying efficiencies. Additionally, when used with space sharing [103, 142], each job can be run with many other jobs (again with varying efficiencies). We assume that dependencies that specify when jobs are runnable are handled by a separate DAG scheduler. This is standard in systems such as Spark and Hadoop [145]. Such cross-job "when can job X run" dependencies are not under the purview of the resource schedulers considered in this chapter, which try to determine how resources should be shared among already *runnable* jobs only.

In the traffic engineering scenario, as we have already seen in Chapter 3, the clients are commodities, each resource is a network link between two sites in the WAN, and each commodity typically requests a small fraction of the total available capacity. In load balancing, the clients are data shards, the resources are servers, and each shard can be handled by a small fraction of the total number of servers available in the system.

6.3 Partitioned Optimization Problems

Granular resource allocation problems can be split into sub-problems, where each sub-problem has a subset of the clients and resources in the full allocation problem. We leverage the large number of clients and resources to randomly partition clients and resources into sub-problems; this procedure yields high-quality allocations due to the law of large

numbers. We call this technique Partitioned Optimization Problems (or POP for short). In the rest of this section, we describe the intuition, procedure, and benefits of POP.

6.3.1 Intuition

Optimization problems for large systems take a long time to solve in part because they have many variables. For example, consider an optimization problem that involves scheduling n jobs on m cloud VMs. Each VM has varying amounts of resources (e.g., CPU cores, GPUs, and RAM). To express the possibility of any job being assigned to any VM, an $n \times m$ matrix of variables would be needed; for 10^4 jobs and 10^4 VMs, the problem has 10^8 variables. Contemporary solvers often take hours to solve such problems, although the exact runtime depends on problem properties such as sparsity [136].

We can achieve much faster allocation computation times by decomposing the problem; for example, the problem of scheduling 10^3 jobs on 10^3 VMs ($100\times$ fewer variables) is much more tractable. This procedure of breaking up the larger problem into sub-problems *reduces the search space* explored by the solver, since interactions between all combinations of clients and resources are no longer considered. Instead, only combinations of *subsets* of clients and resources are considered, which reduces runtime but also can reduce the quality of the allocation. In light of this, the interaction between clients and resources needs to be considered carefully to take into account the many global constraints in the original problem, as well as the objective (e.g., fairness). We find that on large granular resource allocation problems, splitting clients *randomly* and assigning an equal number of resources among sub-problems reduces the search space of feasible solutions that needs to be considered by solvers, while still ensuring that *some* high-quality feasible points are in the explored search space. This is the main intuition that allows POP to be effective, returning allocations of similar quality as the original formulation but faster.

6.3.2 Procedure for Granular Problems

The first step of POP is to **`partition`** larger allocation problems into smaller allocation sub-problems. The type of partitioning allowed is dependent on the objective and constraints of the allocation problem, and has implications on the runtime speedups and

Algorithm 3 POP Procedure.

Input: Clients and their attributes $X = [x_1, x_2, \ldots, x_n]$, resources and their attributes $Y = [y_1, y_2, \ldots, y_m]$, function to compute allocations GET_ALLOCATION : $(X, Y) \rightarrow A$, number of sub-problems l, (optional) splitting attribute s, (optional) ratio of extra virtual clients allowed t.
Return: Allocation for all n clients, A.

// Optional: make the problem granular if it is not already.
$X' = \text{SPLIT\_CLIENTS}(X, s, t)$, $Y' = \text{SPLIT\_RESOURCES}(Y)$

// This is the **`partition`** *step.*
$[X'_1, X'_2, \ldots, X'_l], [Y'_1, Y'_2, \ldots, Y'_l] =$ **`partition`**(X', Y', l)

// This is the **`map`** *step, can be performed in parallel.*
for i in `range`(l) **do**
  $A_i = \text{GET\_ALLOCATION}(X'_i, Y'_i)$
end for

// This is the **`reduce`** *step; allocations A_i are combined.*
$A = \text{COALESCE}([A_1, A_2, \ldots, A_l])$

quality of the returned allocation. We can then re-use the map-reduce API [48, 145] (or divide-and-conquer): each of these sub-problems can be solved in parallel (**`map`** step), and then allocations from the sub-problems can be reconciled into a larger allocation for the entire problem (**`reduce`** step). We show pseudocode for this in Algorithm 3.

The partitioning step affects the runtime, the reconciliation complexity, and ultimately the quality of the final allocation. One straightforward approach that we explore in this chapter is to divide *both* clients (e.g., jobs, shards) and resources (e.g., servers) randomly into *sub-systems*, as shown in the top half of Figure 6.2. We find that this partitioning scheme is effective even when clients have attributes with skew (e.g., jobs in a shared cluster with various priority levels, or data shards in query load balancing with different loads). Low-quality allocations can also result from clients having vastly different utilities with different resources. For example, a resource could be a network link between two sites in a WAN. A commodity might *have* to use this link to send traffic between these two sites. This chapter shows how client and resource splitting (§6.3.3) can be used to transform some

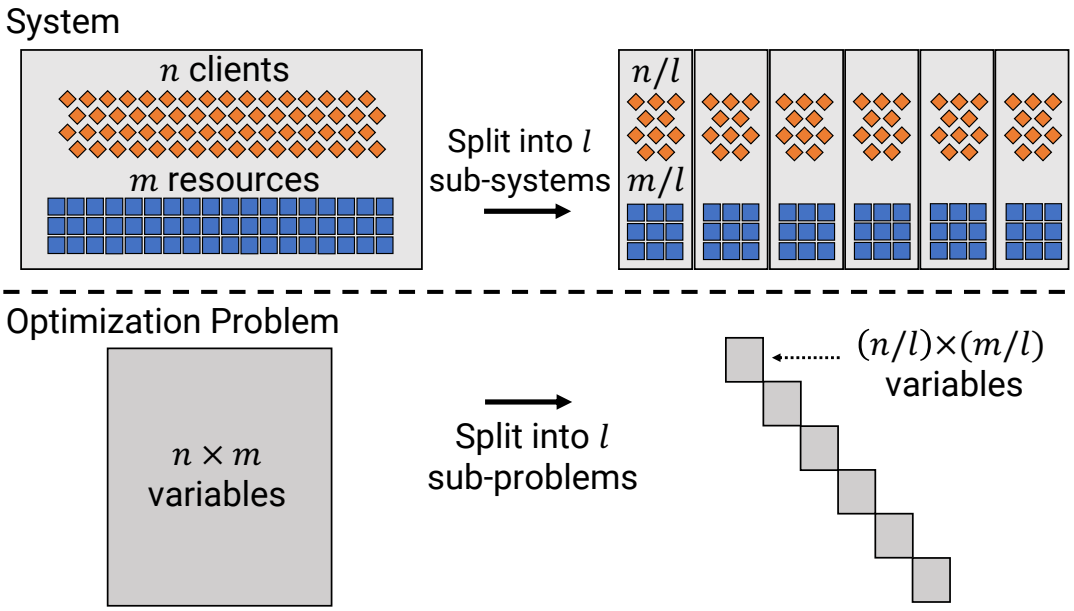


Figure 6.2: POP partitions the system to reduce the number of optimization problem variables. For a problem where the number of variables is the number of clients times the number of resources, dividing clients and resources evenly among l sub-problems reduces the number of variables in each sub-problem by l^2.

of these "hard" problems into a form that is then amenable to *random partitioning*. Other broad partitioning strategies can also be used depending on problem structure (e.g., assign all "geographically close" clients and resources to the same sub-problem), but these are out of the scope of this chapter. With random partitioning, the `reduce` step is cheap, as simply concatenating sub-system allocations yields a feasible allocation to the original problem.

6.3.3 Transformations to Granularize Problems

In some cases, it might not be possible to either return an allocation that is feasible or high quality by merely assigning each client and resource to sub-problems at random when using the POP procedure. Skewed workloads with heavy tails are common in practice [133]. As an example, consider a query load balancing problem where we try to assign shards containing various keys to compute servers: our goal is to spread load evenly amongst the available servers, which can be formulated as a mixed-integer linear program (§6.5). In such a setting, it is common for single shards to be *hot*: for example, Taylor Swift's

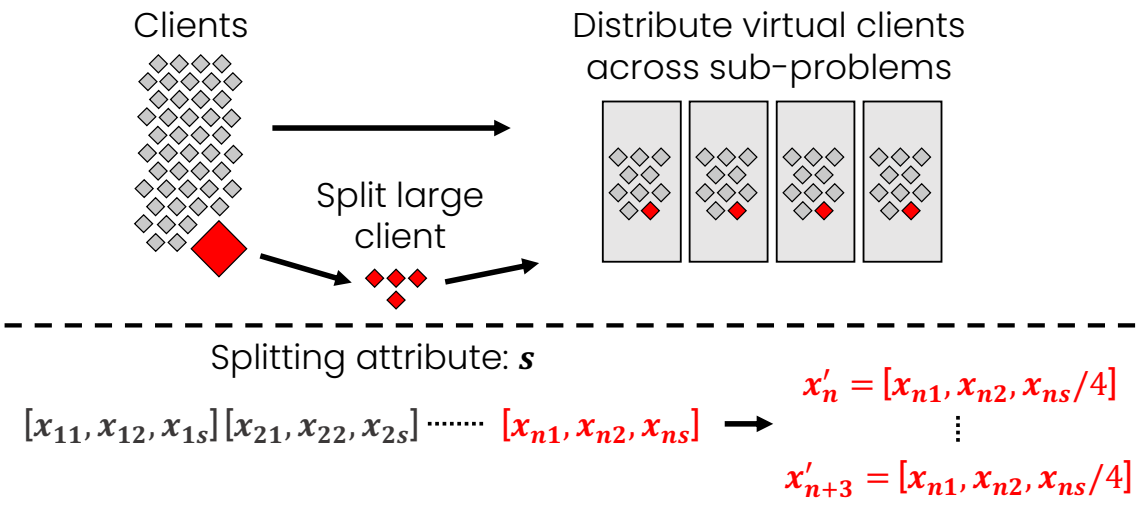


Figure 6.3: Client splitting, where we *granularize* non-granular problems by splitting clients based on a splitting attribute s.

Twitter account receives much more request traffic compared to the average Twitter user. In light of these hot shards, it might not be possible to assign shards to individual sub-problems and obtain sub-problems with input distributions similar to the original problem (and consequently leading to either an infeasible or poor-quality allocation). To transform these into granular problems, we propose an algorithm to *split* variables for clients and resources across several sub-problems.

Client Splitting. We require the user to specify the client attribute that specifies resource demand and can be split across several sub-problems; all other attributes are copied over without change. In the load balancing example, where clients are data shards and attributes include shard load and memory size, the *splitting attribute* is the shard load. In the traffic engineering example, the splitting attribute is the commodity's traffic demand. Given this splitting attribute, we then construct a priority queue (heap) of the corresponding attribute values for all clients. Given a threshold t (t is typically a number less than 1) on the maximum number of extra *virtual* clients allowed, we pop and split variables off the queue, and then push the new variables back into the queue. Each split reduces the value of the splitting attribute of the popped variable by a factor of 2. Importantly, each split maintains the feasibility invariant: the coalesced allocation across virtual clients will still be feasible

Algorithm 4 Client Splitting Algorithm.

Input: Inputs $X = [x_1, x_2, \ldots, x_n]$, splitting attributes s, ratio of extra virtual clients t allowed.
Return: Mapping from real to virtual clients $\{x_i \rightarrow [x'_j]\}$.
Initialize queue $\leftarrow$ MAX_HEAP(), mapping $\leftarrow \{\}$.
For all $i \in \{1, 2, \ldots, n\}$, queue.PUSH($x_i.s, x_i$).

while len(queue) $\leq (1+t) \cdot n$ **do**
  $x_{\text{max}} =$ queue.POP()
  Split x_{max} by attribute s into two copies x^1_{max} and x^2_{max} ($x^1_{\text{max}}.s, x^2_{\text{max}}.s = x_{\text{max}}.s/2$).

  UPDATE_MAPPING($x_{\text{max}}, [x^1_{\text{max}}, x^2_{\text{max}}]$)
  queue.PUSH($x^1_{\text{max}}.s, x^1_{\text{max}}$), queue.PUSH($x^2_{\text{max}}.s, x^2_{\text{max}}$)
end while

(since the total sum of splitting attribute values remains the same). By reducing the value of the splitting attribute, client splitting breaks down large clients into a collection of smaller clients with *equivalent* total demand. The runtime of this algorithm is $O(n \log n)$, where n is the number of clients, which is cheap compared to the runtime of allocation computation in each sub-problem. Algorithm 4 shows pseudocode, and the procedure is illustrated in Figure 6.3. Empirically, we found that most problems are granular enough for POP to work well with 0 split clients. Client splitting does not adversely impact allocation quality, but can increase runtime. The hardest problems in our experiments required $t = 0.75$. The optimal value of t is problem-specific and it is possible that users may have to dynamically adapt t to get the best performance from POP; however, in all of the considered production use-cases in our experiments, we found that small values of t that worked well for historical problem instances continue to work well on future problem instances.

Resource Splitting. If a client has to use a particular resource to make progress, POP will not work out of the box, since randomly partitioning clients and resources into sub-problems might result in a partitioning where the client is not matched with its preferred resource. In such cases, each resource can be split into l "virtual" resources (where l is the number of sub-problems). Each virtual resource has $l\times$ lower capacity, and is assigned to a different sub-problem. By ensuring that each virtual resource has lower capacity, we

ensure that the final coalesced allocation is still feasible.

Client and resource splitting are not always applicable. For example, resource splitting cannot be used easily if the allocation problem's objective depends on whether a resource is used or not (e.g., an allocation problem that tries to minimize the number of resources used). Similarly, client splitting cannot be used easily for problems which take into account interactions between multiple clients sharing a resource.

The resulting allocation problem after these transformation steps can be granular; if so, we can use POP to solve it. After the `partition` step, we obtain allocations for each virtual variable in the problem. Allocations assigned to virtual variables corresponding to a single client need to be summed to obtain the final allocation. We show how this can be incorporated into the full POP procedure in Algorithm 3.

6.3.4 Benefits of POP

POP has several desirable properties:

- **Simplicity:** Users do not need to design new heuristics from scratch to scale up to larger problem sizes, and can reuse their original problem formulations.
- **Generality across domains and solvers:** POP can be used to accelerate allocation computations for many different types of problem formulations across domains. POP also easily integrates with different solvers.
- **Applicability to different types of objectives:** POP can be applied for a broad class of objectives, such as minimizing total makespan, maximizing minimum fairness [67], or maximizing proportional fairness [13].
- **Composability:** POP can be used for any granular allocation problem in an outer loop as a simplifying step; existing heuristics or approximation algorithms can then be used to solve the resulting sub-problems.
- **Tunability:** The number of sub-problems is a knob for trading off between allocation quality and runtime.

6.4 Case Studies of Applying POP

In this section, we describe two other resource allocation problems that are formulated as optimization problems: scheduling of jobs on clusters with possibly heterogeneous resources [103] and query load balancing [46, 127, 131]. We show the full *exact* problem formulations presented in the corresponding papers, and then explain how POP can be used to compute high-quality allocations faster. We also present some examples of problems which are not granular and out of scope for POP.

6.4.1 Resource Allocation for Heterogeneous Clusters

We first discuss the optimization problem formulations used in Gavel, which supports a range of complex objectives. These can be accelerated using POP since these problems are granular, i.e., meet the conditions in §6.2.

Gavel [103] is a cluster scheduler that assigns cluster resources to jobs while optimizing various multi-job objectives (e.g., fairness, makespan, cost). Gavel assumes that jobs can be time sliced onto the available heterogeneous resources, and decides what fractions of time each job should spend on each resource type by solving an optimization problem. Optimizing these objectives can be computationally expensive when scaled to 1000s of jobs, especially with "space sharing" (jobs execute concurrently on the same resource), which requires variables for every *pair* of runnable jobs.

Allocation problems in Gavel are expressed as optimization problems in terms of a quantity called *effective throughput*: the throughput a job observes when given a resource mix according to an allocation A, computed as:

$$\text{throughput}(\text{job } j, \text{allocation } A) = \sum_i T_{ji} \cdot A_{ji}.$$

T_{ji} is the raw throughput of job j on resource type i. In Gavel, vanilla heterogeneity-aware allocations A_{ji} are assigned to each combination of job j and GPU type i. A_{ji} represents the fraction of wall-clock time that a job j should spend on the GPU type i. We now show formulations for three objectives.

Max-Min Fairness. The Least Attained Service policy [67] tries to give each job an equal resource share of the cluster. The heterogeneity-aware version of this policy can be expressed as a max-min optimization problem over all active jobs in the cluster. We assume that each job j has fair-share weight w_j and requests z_j GPUs. Then, to take into account the impact of moving a job between GPU types, we find the max-min allocation of normalized effective throughputs:

$$\text{Maximize}_A \min_j \frac{1}{w_j} \frac{\text{throughput}(j, A)}{\text{throughput}(j, A^{\text{equal}})} \cdot z_j.$$

A^{equal} is the allocation given to job j assuming it receives equal time share on each worker type in the cluster. We also need to specify constraints to ensure that jobs and the cluster are not over-provisioned (e.g., total GPU allocation time does not exceed the total number of GPUs):

$$\begin{aligned} 0 \leq A_{ji} \leq 1 \quad & \forall (j, i) \\ \textstyle\sum_i A_{ji} \leq 1 \quad & \forall j \\ \textstyle\sum_j A_{ji} \cdot z_j \leq \text{num\_workers}_i \quad & \forall i \end{aligned}$$

The above formulation can be extended to consider space sharing [103, 142], where multiple jobs execute concurrently on the GPU to improve GPU utilization, by only changing the way effective throughput is computed; see the Gavel paper [103] for details.

Proportional Fairness. Proportional fairness [13] tries to maximize total utilization while still maintaining some minimum level of service for each user (in this case, job). Proportional fairness for GPU cluster scheduling can be formulated as the following convex optimization problem:

$$\text{Maximize}_A \sum_j \log(\text{throughput}(j, A)).$$

Constraints are the same as before. Per-job weights and other extensions are also possible (the above objective can be interpreted as a sum of utilities, i.e., $\text{Maximize}_A \sum_i U_i(A_i)$).

Minimize Makespan. We can also minimize makespan (the time taken by a collection of jobs to complete) using a similar optimization problem framework. Let num_steps_j be the number of iterations remaining to train job j. The makespan can then be computed as the maximum of the durations of all active jobs; the duration of job j is just the ratio of the number of iterations to $\text{throughput}(j, A)$. Mathematically, this can be written as follows using the same above constraints:

$$\text{Minimize}_A \max_j \frac{\text{num\_steps}_j}{\text{throughput}(j, A)}.$$

Using POP. We can use POP on these cluster scheduling problems by partitioning the full set of jobs into job subsets, and the cluster into sub-clusters. Each sub-cluster has an equal number of resources (GPUs of each type), and jobs are partitioned randomly into the job subsets. The POP solution is feasible by construction. Since the cluster has multiple resources of each type (e.g., GPU of specific generation), the problem is granular by default, and does not require additional transformations to be made granular. Additionally, even when allowing job colocation (using space sharing), jobs can make progress colocated with many other jobs.

6.5 Query Load Balancing

Systems like Accordion [127], E-Store [131], and Kairos [46] need to determine how to place data items in a distributed store to spread load across available servers.

We consider the problem of load balancing data shards (collections of data items). This is similar to the single-tier load balancer in E-Store, but acting on collections of data items instead of individual tuples. The objective is to minimize shard movement across servers as load changes, while constraining the load on each server to be within a tolerance ϵ of average system load L. Each shard i has load l_i and memory footprint f_i. Each server j has a memory capacity of memory_j that restricts the number of shards it can host. The initial placement of shards is given by a matrix T, where $T_{ij} = 1$ if partition i is on server j. A is a shard-to-server map, where A_{ij} is the fraction of queries on partition i served by j, and $A'_{ij} = 1$ if $A_{ij} > 0$, 0 otherwise. Finding the balanced shard-to-server map that minimizes

data movement can then be formulated as a mixed-integer linear program:

$$\text{Minimize}_A \sum_i \sum_j (1 - T_{ij}) A'_{ij} f_i.$$

Subject to the constraints:

$$\begin{aligned} L - \epsilon \leq \textstyle\sum_i A_{ij} l_i \leq L + \epsilon \quad & \forall j \\ \textstyle\sum_j A_{ij} = 1 \quad & \forall i \\ \textstyle\sum_i A'_{ij} f_i \leq \text{memory}_j \quad & \forall j \\ A_{ij} < A'_{ij} \leq A_{ij} + 1 \quad & \forall (i,j) \end{aligned}$$

Using POP. The load balancing problem can be accelerated using POP by dividing the shard set and server cluster into shard subsets and server sub-clusters, while ensuring that each shard subset has the same total load.

6.6 When is POP Not Applicable?

Although POP can be used on a number of different resource allocation problems, it cannot be used for *all* possible problem formulations. Here, we present a few examples of resource allocation problems where POP with random partitioning *cannot* be used.

Capacitated Facility Location. The capacitated facility location problem tries to minimize the cost of satisfying users' demand given a set of processing facilities. Each facility has a processing capacity, and also a "leasing cost" if used at all (if a facility is not processing any demand, it has a leasing cost of 0). The cost of processing some demand by a facility is proportional to the distance of the facility from the user. Problems where a user is only close to a single facility are not amenable to POP and violate **condition 3** in the definition of granularity: partitionings of the problem where the user is not placed into the same sub-problem with the facility closest to them would lead to a low-quality allocation. Additionally, resource splitting cannot be used to make the problem granular, since the objective explicitly takes into account whether facilities are used or not, and creating multiple

variables for a single `(client,resource)` pair would require additional constraints *across* sub-problems. More generally, resource allocation problems where clients prefer one resource over all other available resources by a large amount are a poor fit for POP unless resource splitting can be used.

Global Rescheduling with Plan-Ahead. TetriSched [134] is a scheduler that can take into account upcoming resource reservations when deciding how to allocate resources to jobs. TetriSched allows preferences to be specified declaratively (e.g., a job comes in at a specific start time and needs to be completed by a specific end time). These preferences are then compiled into a mixed-integer linear program (MILP). These MILPs can be accelerated using POP by dividing the jobs and resources into job and resource subsets, and solving each sub-problem independently. However, TetriSched also supports combinatorial constraints, such as "a particular set of k jobs must use the same resource", which cannot be supported by POP without smarter partitioning algorithms.

6.7 Analysis

The effectiveness of POP is directly tied to how clients and resources are partitioned across sub-problems. In this section, we consider a simple resource allocation problem and prove that the probability of a large optimality gap with the POP procedure and random partitioning is low, discuss how POP relates to *primal decomposition* (a technique used in convex optimization to decompose certain types of optimization problems), and also note the expected runtime benefits.

6.7.1 Theoretical Analysis for a Simple Problem

In settings with large numbers of clients, POP with random partitioning works well. In this section, we consider a simplified allocation problem and compute an upper bound on the probability that POP (using l sub-problems) with random partitioning results in a low-quality allocation.

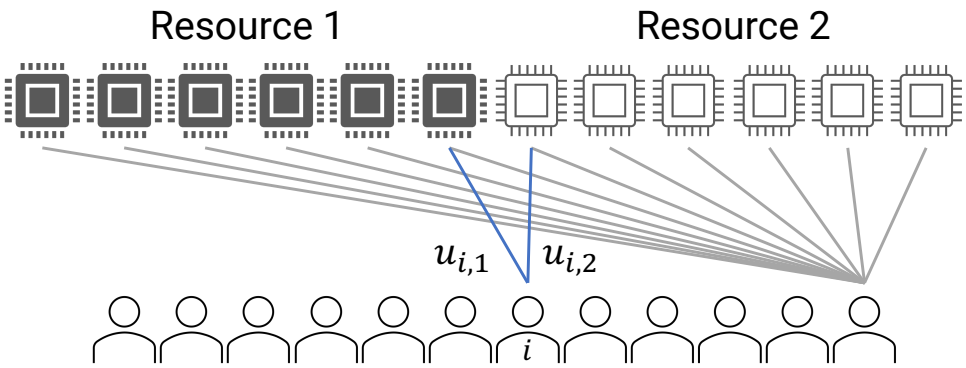


Figure 6.4: Simple partitioning problem where jobs are assigned servers (or resources). Each job i derives utility $u_{i,1}$ from resource 1 and $u_{i,2}$ from resource 2.

The allocation problem we consider assigns servers to jobs. We assume that the problem has the following properties:

- n jobs and servers. Each job is allocated a single server.
- r distinct server types (equal number of each type).
- Job i has utility $u_{i,s}$ on resource type s.
- The largest difference in utility for any job across any two servers is u_{maxgap}.

A job is "type-s" if it achieves highest utility on a type-s server. With two server types, we have type-1 and type-2 jobs (shown in Figure 6.4).

The objective of this problem is to maximize the overall utility of the allocation, defined as the sum of every job's utility on its assigned server.

Now, if we use POP to solve this problem, we would equally partition servers of each type into sub-clusters, randomly assign jobs to sub-clusters, and then solve assignment problems separately for each sub-cluster. We wish to answer the following questions in this regime:

1. What is the optimality gap of the solution using the POP procedure (with respect to the optimal solution for the full problem)?
2. How do the values of n, r, u_{maxgap}, and l affect this optimality gap?

One way to quantify the optimality gap is to count the number of "misplaced" jobs in each sub-problem (e.g., type-1 jobs that are not assigned "resource 1" because there were too many other type-1 jobs in the relevant sub-problem). Define $q_{s,t}$ to be the number of type-s resources that are misplaced in sub-problem t. The distance from optimal utility, i.e., optimality gap, is bounded by the product of this number and u_{maxgap} added across all resource types and sub-problems:

$$\text{Optimality gap} \leq \sum_{s=1}^{r} \sum_{t=1}^{l} q_{s,t} u_{\text{maxgap}} \tag{6.1}$$

We note that this is a loose bound for the gap, since jobs with large resource utility gaps would be allocated their optimal resource even within a sub-problem.

To quantify the performance gap between POP and optimal solutions, we now need a sense of how big $q_{s,t}$ can be in practice. We walk through the full derivation of a bound on the probability that the optimality gap exceeds a given value in the Appendix, but briefly sketch it here. The random assignment of all type-s jobs to sub-problems can be interpreted as Bernoulli trials where the probability that any given type-r job is placed in a given sub-problem is $1/l$. We then use a classical Chernoff bound [100] to compute the probability that each $q_{s,t}$ exceeds a fraction δ of its expected value (n/rl). We can combine these across all job types and sub-problems using the union bound to find an upper limit on the probability that the total number of misplaced jobs exceeds δn. This allows us to bound the distance of a randomly-partitioned POP allocation from optimal utility by $\delta u_{\text{maxgap}} n$:

$$\Pr\left[U(\Gamma^*) - U(\Gamma^{\text{POP}}) \geq \delta u_{\text{maxgap}} n\right] \leq rl \exp\left(\frac{-\delta^2 n}{(2+\delta)rl}\right) \tag{6.2}$$

where Γ^* is an optimal allocation, Γ^{POP} is the allocation returned by the POP procedure, and $U() : \Gamma \rightarrow u$ is a function that maps an allocation Γ to a scalar value (the utility).

Equation 6.2 defines the relationship between the problem parameters (n, r, u_{maxgap} and l) and the probability that the optimality gap exceeds a given fraction δ of the worst-case gap if every job is allocated its worst resource ($u_{\text{maxgap}} n$). Concretely, the probability decays exponentially with n; as the problem gets larger, the probability of having a large optimality gap becomes very small. The probability also decays exponentially with δ^2. On the other

hand, the probability of a large optimality gap increases as r, l, and u_{maxgap} increase; this is to be expected, as having many sub-problems and many resource types increases problem heterogeneity and makes it more likely for a random partitioning to lead to misplaced jobs and a lower-quality allocation.

To put this bound into perspective, consider a large cluster with 1 million jobs, l = 10 sub-problems, and r = 4 resource types of equal amounts ($n/rl = 25,000$); the probability that more than 3% of jobs are not allocated their optimal resource is upper bounded by 0.000614.

To summarize, the bound given in Equation 6.2 for a simple allocation problem gives insight as to why POP works well empirically for more complex granular resource allocation problems like those described in §6.4.

6.7.2 Relationship to Primal Decomposition

For many problems, such as when the objective function is separable and convex (that is, the objective can be expressed in the form "Maximize $U(A) = \sum_i U_i(A_i)$" with per-job utility functions U_i), POP can be interpreted as the first iteration of primal decomposition, a well-known method from convex optimization [30]. Primal decomposition is an iterative technique; for a resource allocation problem, it works by decomposing the large problem into several smaller allocation problems, each with a subset of clients and resources. In each iteration, every sub-problem is solved individually, and then the dual variables of each sub-problem are used to determine how to shift resources between the sub-problems; those found to be relatively resource-starved are given more resources from other sub-problems for the next iteration.

Like many other techniques from the optimization literature, primal decomposition works for a restricted set of problems, namely those with separable objectives and certain types of constraints (see Boyd et al. [30]). These restrictions come into effect during the resource-shifting phase prior to subsequent iterations. For a "well-partitioned" problem with a separable objective (i.e., each sub-problem has sufficient resources), one iteration of primal decomposition is often sufficient and resource shifting is not required [30]. Primal decomposition and POP are thus equivalent for these problems, explaining why POP can

produce a high-quality allocation efficiently. However, this explanation does not apply to other problems where primal decomposition cannot be used (e.g., non-convex problems, such as the MILP used in the load balancing problem from §6.5), even though we found POP to still be effective in such regimes.

6.7.3 Expected Runtime Benefits

We can estimate the runtime benefits of POP when used with linear programs. Solvers for linear programs have worst-case time complexity of $O(f(n,m)^a)$ ($a \approx 2.373$ [44] in the worst case) where $f(n,m)$ is the number of variables (n clients and m resources) in the problem. If $f(n,m) = n \cdot m$ and both clients and resources are partitioned across l sub-problems, each sub-problem will have $l^2\times$ fewer variables, as illustrated in Figure 6.2. The asymptotic runtime savings are then proportional to l^{2a-1} if each sub-problem is solved serially, and proportional to l^{2a} if solved in parallel, assuming a cheap **`reduce`** step. Some problems have an even larger potential for runtime reduction. For example, if the allocation considers interactions between two jobs on the same resource, then the problem would have n^2m variables, and using POP would lead to a larger runtime speedup (proportional to l^{3a-1} if each sub-problem is solved serially, and proportional to l^{3a} if solved in parallel).

6.8 Implementation

This more general form of POP is easy to implement on top of a number of existing solvers for a variety of different granular allocation problems. The main method that needs to be implemented is **`partition`**, which given a collection of clients and resources, assigns them to sub-problems. The subsequent **`map`** step then involves calling the existing solver routine for the already-written problem formulation on the smaller sub-problem. The **`reduce`** step is similarly simple, and involves concatenating the allocations obtained from each of the sub-problems and summing allocations across virtual clients and resources (when using client and resource splitting).

We implemented POP on top of a number of different solvers (MOSEK using `cvxpy` [14, 50], Gurobi [68], and a custom solver [13] that uses PyTorch [114]) for problems across

diverse domains, in $<$ 20 lines of code in each case. We implemented client splitting in about 100 lines of Python code. Our evaluation code—which includes specific implementations for cluster scheduling and load balancing—is open-sourced on GitHub at https://github.com/stanford-futuredata/POP.

6.9 Evaluation

In this section, we seek to answer the following questions:

1. What is the effect of POP on allocation quality and execution time on granular allocation problems? How does it compare to relevant heuristics?
2. Does POP work across a range of solvers and types of optimization problems?
3. How effective are POP's client and resource splitting optimizations in generating high-quality allocations?
4. How does random partitioning compare to other more sophisticated problem partitioning strategies?

We evaluate POP on problems from two domains:

1. **GPU cluster scheduling**, where we apply POP to solve the optimization problems used in Gavel (§6.4.1), and compare with the greedy Gandiva policy [142].
2. **Shard load balancing** in distributed storage systems, where we apply POP on the problem formulation in §6.5, and compare to a heuristic from E-Store [131].

Where relevant, we integrate POP into systems such as Gavel [103] to measure the end-to-end impact of POP on application performance. Our results span three different cluster scheduling policies (max-min fairness, minimize makespan, and proportional fairness) and one load balancing policy (minimize number of shard transfers as load changes).

We first present end-to-end experiments, then present some microbenchmarks that examine the impact of various algorithmic contributions in POP.

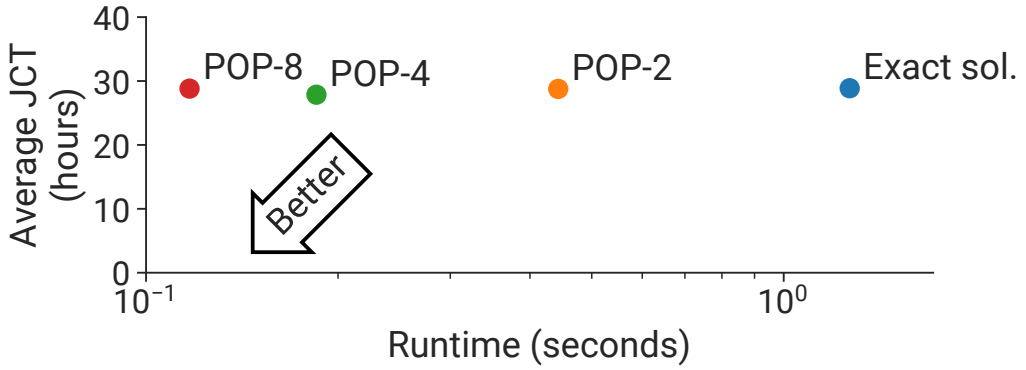


Figure 6.5: Results for the max-min fairness policy (with space sharing) for cluster scheduling for the formulation shown in §6.4.1 ("Exact sol.") and its POP variants. POP-l uses l sub-problems.

6.9.1 End-to-End Results

We first demonstrate POP's end-to-end effectiveness on various problems. We compare to approaches based on allocation quality, and time needed to compute the allocation; the runtime for POP includes the runtime for solving the optimization problems for sub-problems. In all of our experiments, "Exact sol." is the original unpartitioned problem formulation and solver used by the reference system (e.g., Gavel for cluster scheduling). We believe this is a fair baseline since it represents what people use today if using optimization problem formulations for resource allocation. We use the same evaluation methodology as related work. The total number of threads given to solvers for our baselines and POP are the *same*. If l sub-problems are solved in parallel when using POP, each sub-problem uses $1/l$ of the number of threads. We also present heuristics where relevant. Unfortunately, not every problem has a state-of-the-art heuristic. For example, it is not clear how to use a heuristic to solve for an approximate proportionally-fair allocation. We explicitly note when we use client or resource splitting.

Cluster Scheduling

We used POP to accelerate various cluster scheduling policies supported by Gavel [103]. We then used these POP-ped policies in Gavel's full simulator[2] to measure the impact of

[2]The Gavel paper [103] shows that its simulator demonstrates performance very similar to behavior on the physical cluster.

POP on end-to-end metrics of interest, like average job completion time and makespan for real traces. The traces and methodology used are identical to those used in Gavel.

Max-Min Fairness. We show the trade-off between runtime and allocation quality for the max-min fairness policy with space sharing on a large problem (2048^2 job pairs on a 1536-GPU cluster) in Figure 6.1 (in the introduction). POP leads to an extremely small change in the average effective throughputs across all jobs ($< 1\%$), with a $22.7\times$ improvement in runtime. Gandiva [142], on the other hand, uses a heuristic to assign resources to job pairs, resulting in $1.9\times$ worse allocation quality.

We unfortunately could not run end-to-end simulations for such large problem sizes: the simulation involves running thousands of allocation problems, since an allocation problem needs to be solved every time a new job arrives at the cluster or an old job completes. This would take months to run at scale by virtue of the number of problems that need to be solved and the time taken for each problem. Instead, we show full simulation results on more moderate problem sizes. These experiments involve dynamic changes: the full simulation involves new jobs coming in and old jobs completing, and consequently the set of jobs is not static.

We ran experiments with 96 GPUs (32 V100, P100, and K80 GPUs). The original heterogeneity-aware Least Attained Service policy without space sharing has a small number of variables (on the order of hundreds). Even on such smaller problem sizes, the quality of allocation with POP is high, with only up to a 5% drop in average JCT (not pictured).

Figure 6.5 shows the average JCT of the original Least Attained Service policy from §6.4.1, with space sharing, along with three POP-ified versions using 2, 4, and 8 sub-problems. With space sharing, the number of variables scales quadratically with the number of jobs: this leads to a performance speedup of $11\times$ with $l = 8$ compared to the full problem formulation, and similar average JCT.

We see similar behavior for max-min fairness policies when clients have more attributes (e.g., different priority levels). Average JCTs are almost identical when jobs request multiple GPUs, and increase by 5% for high-priority jobs in workloads containing a mix of low- and high-priority jobs, using the Gavel simulator as before.

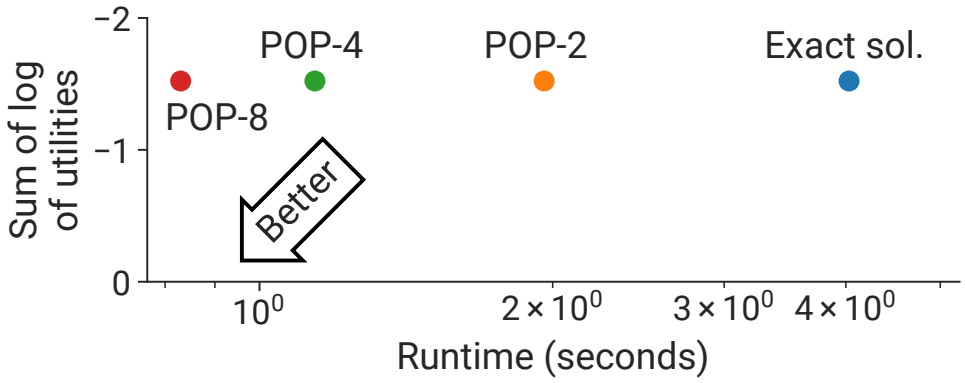


Figure 6.6: Results for the proportional fairness policy for cluster scheduling for the formulation shown in §6.4.1 ("Exact sol.") and its POP variants. POP-l uses l sub-problems.

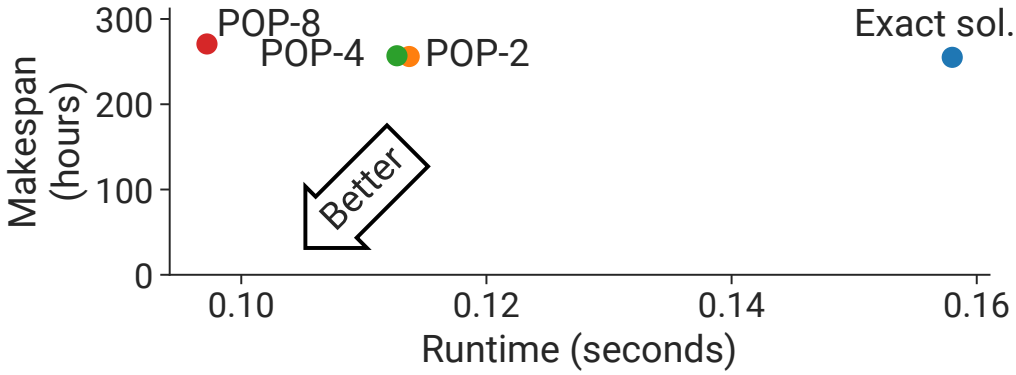


Figure 6.7: Results for the minimize makespan policy for cluster scheduling for the formulation shown in §6.4.1 ("Exact sol.") and its POP variants. POP-l uses l sub-problems.

Proportional Fairness. We ran a simple experiment with the proportional fairness policy with 10^6 jobs and a similar number of resources. Figure 6.6 shows POP combined with a proportional fairness policy. This allocation problem is a general convex optimization problem (not a linear program), with a sum-of-$\log$ objective. For this problem, we implement POP on top of a custom solver [13] that runs an order of magnitude faster than commercial solvers for this particular problem formulation. We see strong scaling performance as we increase the number of sub-problems (4.9$\times$ reduction in runtime with 8 sub-problems), with an extremely small optimality gap (7×10^{-5}).

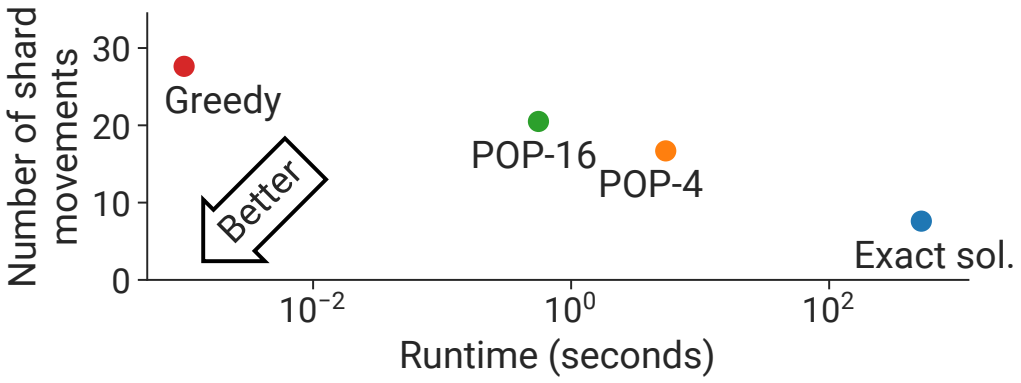


Figure 6.8: Results for the minimize shard movement policy for load balancing for the formulation shown in §6.5 ("Exact sol.") and its POP variants. We also compare to a greedy heuristic ("Greedy"). POP-l uses l sub-problems.

Minimize Makespan. Figure 6.7 shows the makespan of variants of the minimize makespan policy. This policy again is a simple linear program with number of variables linear in the number of jobs and resource types. Consequently, the runtime improvements are lower ($1.6\times$), but the end-to-end makespan over the trace is nearly identical.

Load Balancing

In Figure 6.8, we evaluate POP on a load balancing problem. In the problem, we have 1024 shards of data each assigned to exactly one of 64 servers. Each round, we receive the query load of each shard and compute a new assignment of shards to servers such that each server has approximately (within 5%) the same amount of load across its shards but the number of shard movements is minimized. We examine the performance of POP with various numbers of sub-problems and compare it to the original optimization problem (§6.5) and a greedy heuristic algorithm from E-Store [131]. For each system, we run 100 rounds of the problem. In each round, we generate a new load distribution and rerun the load balancing algorithm. We report the average runtime and number of shard movements across these rounds.

We find that POP improves the runtime over the original problem by two orders of magnitude, while outperforming the greedy heuristic. The exponential scaling of MILP solvers restricted us to smaller problem sizes for the purpose of comparing against the

optimal solution. Since shard movements are stateful (previous round's solution is initial state for current round), we added an extra step to re-balance the aggregate load in the relatively small sub-problems, requiring a few extra shard movements. As l increases, the number of sub-problems and thus the number of these movements also increases, which is why POP-l does worse as l increases. This becomes less of an issue for larger problem sizes where random allocations are likely to remain balanced.

6.10 Related Work and Discussion

In this section, we discuss other systems that use optimization problems to allocate resources. We also comment on general efforts to accelerate solving large optimization problems and how POP fits into this body of work.

Optimization Problems in Systems. A number of systems besides the ones discussed in §6.4 use optimization problem formulations to solve resource allocation problems.

TetriSched [134] is a cluster scheduler that is able to leverage runtime predictions and deadline information (provided as input to the system) to make smarter near-term decisions on how jobs should be allocated resources, while also providing room for uncertainty from unknown future job submissions. Preferences in resource space-time can be expressed in a new DSL called STRL; these are then compiled down to a mixed-integer linear program (MILP) whose solution describes when and how jobs should be executed.

RAS [106] is a capacity reservation system that manages the allocation of servers to clients within a datacenter region, while taking into account failures, resource heterogeneity, and maintenance schedules. RAS formulates problems as MILPs which are solved hourly.

DCM [130] makes it easier to implement various cluster management policies (e.g., ensure containers have enough of a particular resource, or two containers are not placed in the same rack) by having users specify cluster manager behavior declaratively through SQL queries written over cluster state maintained in a relational database. Similar to TetriSched, these queries are then compiled down to an optimization problem that can be solved by constraint solvers, such as CP-SAT [2]. DCM supports affinity and anti-affinity constraints.

Quincy [75] and Firmament [65] are centralized datacenter schedulers that use efficient min-cost max-flow (MCMF) based optimization to scale up to large clusters. Another approach to quickly find good solutions is through variable aggregation [95], where a group of similar variables is represented with a single meta-variable, and then an optimization problem over the meta-variables is solved. The meta-solution can be used to derive a solution to the original problem.

SketchRefine [32] uses a similar approach to accelerate MILPs for "packaging" queries in a database, which handle constraints and preferences over *answer sets*. It uses a quadtree-based partitioning algorithm to group tuples (rows in the relation) into tuple subsets, and then uses an iterative reconciliation procedure to convert initial per-group solutions into a global solution. SketchRefine's partitioning step can be expensive (on the order of minutes), since it is meant to be run over a fixed tuple set. While it is not clear how to extend SketchRefine to resource allocation problems, which reason about interactions between clients and resources, it offers another way to quickly compute good solutions to certain types of large optimization problems in systems.

More Efficient Solving. The optimization community has developed various methods for scaling optimization solvers to handle large problems. Fundamentally, these approaches rely strictly on identifying and then exploiting certain mathematical structures (if they exist) within the problem to extract parallelism; they make no domain-aware assumptions about the underlying problem. For example, Benders' decomposition [62, 119] only applies to problems that exhibit a block-diagonal structure; ADMM [28, 110] has been applied to select classes of convex problems, and Dantzig-Wolfe decomposition [47], while more broadly applicable, offers no speedup guarantee. This poses a significant limitation when applying these methods to real-world systems, which often do not meet their criteria or would need mathematical analysis to determine if this structure exists.

As mentioned in §6.7.2, POP can be interpreted as the first iteration of primal decomposition for optimization problems with separable objectives and certain types of coupled constraints [30]. By randomly partitioning large numbers of clients and equally apportioning resources into sub-problems, we found that it is possible to obtain high-quality solutions with a single iteration for a broader set of allocation problem formulations, including

MILPs.

6.11 Summary

In this section, we showed how to extend POP to a number of resource allocation problems in computer systems beyond traffic engineering. These problems are *granular*, which means they can be partitioned into more tractable sub-problems by randomly assigning clients and resources without significantly sacrificing optimality. POP achieves strong results across a variety of tasks beyond TE, including cluster scheduling and load balancing, with runtime improvements of up to $100\times$ with small optimality gap, outperforming greedy ad-hoc heuristics. We hope that this work motivates fellow researchers to use POP as a simple pre-solving step when solving optimization problems that arise in computer systems.

Chapter 7

Conclusion

How should we route traffic in large-scale wide-area networks, so that we can balance efficiency and optimality? In this dissertation, we have shown that, by intelligently partitioning the traffic engineering problem, we can achieve a better trade-off between these two important goals. We presented two different partitioning schemes—geographic partitioning (NCFlow) and commodity-based partitioning (POP)—that outperform the state of the art, and we illustrated their relative strengths and weaknesses. While these solutions may seem simple at the surface, we've shown that the underlying complexity of these techniques must be addressed to succeed on real-world traffic engineering problems, especially on a wide variety of traffic patterns and topologies.

Undoubtedly, traffic engineering will continue to evolve over the coming years, and we hope that our proposed algorithms serve, at the very least, as a useful reference point in this line of research moving forward. There are a plethora of problems in this domain: failure handling, capacity planning, and traffic forecasting, to name a few. We believe that scalability will continue to be a fundamental challenge, especially as WANs grow and evolve to meet the needs of end users around the world. Moreover, improvements in scalability can have downstream impacts on these secondary and tertiary problems; for example, we showed in Chapter 3 that a fast TE solver can respond quickly to failures in the network, which may be advantageous compared to more proactive failure strategies.

Lastly, we also hope that our work finds an audience beyond the networking community, because the techniques we have described here could potentially provide value beyond

the TE setting. For example, techniques like NCFlow could prove useful in analogous domains, such as social network analysis. Furthermore, in Chapter 6, we discussed how POP can be applied to other resource allocation problems in the computer systems community, such as cluster scheduling and load balancing. We are cautiously optimistic that there are other domains that could also take advantage of POP's simple yet effective approach for allocating resources for large-scale problems.

In the rest of this chapter, we summarize a few of the lessons we have learned in the course of our research, and we examine some of the broader impacts NCFlow and POP have had beyond academia. Finally, we conclude by outlining opportunities for future work.

7.1 Lessons Learned

Combining mathematical approximations with domain-aware insights. Both NCFlow and POP are novel in that they marry mathematical approximations with domain-aware optimizations to produce a better trade-off between optimality and runtime efficiency. In essence, we made deliberate modifications to the underlying TE linear programs to achieve our results: in NCFlow, we created a set of new linear programs, while in POP, we effectively omitted large subsets of variables from the original LP.

From the viewpoint of a theorist, such behavior might be considered cavalier; we are not applying tried-and-true approximation strategies that are common in the theory literature, such as FPTAS. (Fleischer's algorithm is an example of such an approach.) These algorithms are important because they provide a very clear bound on the approximation error (i.e., some ϵ) that is also directly linked to the algorithm's runtime complexity. Similarly, experts in convex optimization may quibble with our decision not to employ traditional methods like Dantzig-Wolfe decomposition, Benders decomposition, or ADMM, all of which provide theoretical guarantees.

However, because these algorithms avoid making stronger assumptions, they are limited in their efficacy, and they do not necessarily realize the ultimate end goal of our research: a better trade-off between optimality and efficiency. In NCFlow, we proposed geographic partitioning *because* of the nature of WAN topologies and the traffic on these WANs. In

POP, we omitted large swathes of the original LP *because* we expected that many commodities (especially those that are located far apart in the topology) will not compete for the same links.

We believe that combining these two strategies—mathematical approximations and systems optimizations, a union of theory and practice—can unlock significant rewards for a host of problems in the computer systems research community. Each of these approaches can pay dividends, but the intersection of the two is underutilized and ripe for more exploration.

Randomization as a path to scalability. When we first proposed POP, many of our colleagues were skeptical that such a simple strategy would be effective for traffic engineering. Although randomization has been applied to TE before (e.g., Valiant Load-Balancing), it waned in popularity with the advent of Software-Defined Networking and the centralized strategies it ushered in.

Why were our peers skeptical? We believe there are multiple reasons: for some, there is a natural aversion to a randomized algorithm for such a mission-critical problem. The pessimist will argue: why would we leave the traffic routes in our datacenter WAN up to chance? The second reason, we believe, is due to a simple feature of our randomized approach: it is quite easy to construct a small adversarial example to illustrate POP's shortcomings, which we demonstrated in Chapter 5. As researchers, we naturally first reason about large-scale problems on a much smaller scale to ensure correctness and build intuition. When one does the same with POP on a WAN topology with only a few nodes, a few links, and a trivial traffic matrix, it becomes glaringly obvious why it could fail.

But therein lies the rub: our goal was never to accelerate TE on small WANs—we only cared about large ones! The oversight here is that a small example does not truly capture POP's effectiveness, because it does not exploit the power of randomized algorithms and the Law of Large Numbers. Instead, one must reason probabilistically and argue that *in expectation* this technique will succeed *because* of how large the problem's inputs are.

From this work, we realized that randomization is a powerful technique for many large-scale problems, and we suspect that there are other problems in other domains that might benefit from this fresh perspective. We encourage researchers in the computer systems

community to experiment with randomized approaches in their own endeavors; the results may surprise you.

7.2 Broader Impact

We have open-sourced both NCFlow and POP on GitHub at https://github.com/stanford-futuredata/pop-ncflow.git. There, one can find implementations of both techniques, as well as the baselines we benchmarked against throughout this dissertation. We also have made public the topologies and synthetic traffic matrices used in our evaluation.

NCFlow was developed in conjunction with several collaborators at Microsoft Research, with the hopes that it would be adopted in production to route traffic in the Azure WAN. While several high-level conversations took place with the Azure Networking team, NCFlow unfortunately never made it to production, due to its limited support for other objectives beyond Maximum Total Flow. POP's support for multiple objectives makes it viable candidate for production systems, and this is ongoing future work.

Additionally, we experimented with POP on additional resource allocation problems at other industrial partners, such as Meta. These collaborations are still ongoing.

7.3 Future Work

As we briefly mentioned in prior chapters, there are a host of interesting follow-up problems that have arisen out of our work on NCFlow and POP. We recap and elaborate on some of those potential extensions below, as well as describe previously unmentioned ones that have piqued our interest and may be worth additional exploration:

Improving NCFlow: As discussed in Chapter 3, NCFlow struggled to find optimal traffic allocations on maximum concurrent flow and min-max link utilization, both of which are "weakest-link" objectives (i.e., the objective value is measured by the worst performing commodity or link). When analyzing NCFlow's shortcomings more carefully on these

objectives, we found that there are opportunities to improve its performance through additional experimentation and modification of our chosen heuristics. For example, a common loss of flow originates from the arbitrary discrepancies between commodities chosen by the source and target clusters in the SrcTargetMax linear program (see Figure 3.4). Imposing a canonical ordering in this LP could mitigate this issue and increase NCFlow's flow allocations. Additionally, we wish to investigate possible recursive clustering strategies with NCFlow, as well as alternative heuristics for allocating flow on inter-cluster edges to address reconciliation issues between clusters.

Improving POP: We covered in Chapter 6 that POP has a broader set of applications beyond traffic engineering, and we illustrated how it draws inspiration from the primal decomposition method, a well-known technique from the convex optimization literature. Specifically, POP represents the first step in primal decomposition, and we wonder: what would happen if we ran it for multiple steps? This would require a shuffling of the commodities between the different sub-problems, which may add some complexity. But doing so could also provide users with a natural progression from POP's almost-optimal solutions to true optimality.

In that same vein, we believe that there are further optimizations that we can apply from the mathematical optimization community. Doubling down on our hypothesis that systems and mathematics must both play a role in tackling these problems (which we previously mentioned in this chapter), we think that a better framing of the optimization problem could lead to other performance improvements and more efficient implementations. For example, optimization researchers have demonstrated how to rewrite the edge formulation LP to be "destination-based," which reduces the total number of variables in the LP by a factor of N, the number of nodes. We could additionally experiment with how the link constraints are defined: for example, expressing them as differentiable functions could unlock Lagrangian methods for solving these LPs, which are quite fast in practice. With the advancements in high-performance computing on GPUs, such computations could be accelerated even further.

Hybrid approaches that leverage both algorithms: As discussed in Chapter 5, NCFlow tends to outperforms POP when a small subset of commodities should dominate the flow allocations to maximize total flow in the WAN, whereas the opposite is true when a more egalitarian approach actually yields the best outcome. This begs the question: is there a way to analyze the WAN topology and traffic matrix *a priori* and identify which technique—NCFlow or POP—is best suited? Such a hybrid scheme would be tremendously appealing to network operators, since it would present them with a can't-lose meta-approach that adapts to the correct situation on its own, without manual intervention. Moreover, the appeal of such a strategy increases as WAN topologies grow in size, since the traffic on such WANs will likely grow in complexity.

Applying NCFlow and POP to other TE problems: Forecasting the growth of the WAN and planning its capacity is a challenging problem for many network operators today; every decision to add a new link to the WAN is extremely costly and cannot be made lightly. (This is especially true for intercontinental links that would be constituted of expensive underwater submarine cables.) Because of the potentially complex interactions between competing traffic interests on the network, most operators rely heavily on simulations to identify future bottlenecks in the WAN. But doing so requires a solver that is both very fast—since so many simulations must be run—*and* very accurate. Both NCFlow and POP could potentially meet this criteria, and we are curious how both techniques would perform in this context. A similar argument exists for failure handling and a litany of other TE problems, and we are excited to see how our algorithms could be leveraged more broadly in traffic engineering research.

We conclude this dissertation by noting that the list above is not exhaustive; in fact, it is anything but. Traffic engineering is a classic problem in computer networking, as old as the Internet itself. And, because of its timelessness, we are confident that there are many problems—some known, but many unknown—that are simply waiting to be discovered and solved. We hope that the contributions we have outlined in this dissertation encourage future researchers to explore this uncharted territory. And above all else, we hope it leads to solutions that push forward our collective understanding of networking, traffic engineering, and computer systems, so that all of us may benefit.

Appendix A

NCFlow

A.1 Properties of NCFlow's flow allocation algorithm

A.1.1 Proof that the algorithm in §3.3.1 meets demand and capacity constraints

Satisfying demand constraints: Commodities whose source and target are in the same cluster are considered by only one instance of MaxClusterFlow; hence, they do not receive more flow than their demands. Specifically, MaxClusterFlow in Figure 3.4 invokes MaxFlow which in turn imposes the demand constraints listed in FeasibleFlow; Equation 2.1.

Commodities whose source and target are in different clusters receive no more flow than their demand due to SrcTargetMax; observe in Figure 3.4 that one of the four constraints in SrcTargetMax explicitly controls the flow for such commodities.

Satisfying edge capacity constraints: We say an edge is local to a cluster if both its incident nodes are within the same cluster. Flow is assigned to a local edge only by the MaxClusterFlow instance of the cluster that contains that edge. Since MaxClusterFlow ultimately invokes FeasibleFlow; by Equation 2.1 a local edge is allocated no more than its capacity.

Edges that are not local receive flow allocation in MaxAggFlow where, as noted in §3.3.1, all of the edges that lie between a pair of clusters are treated as a single edge whose capacity

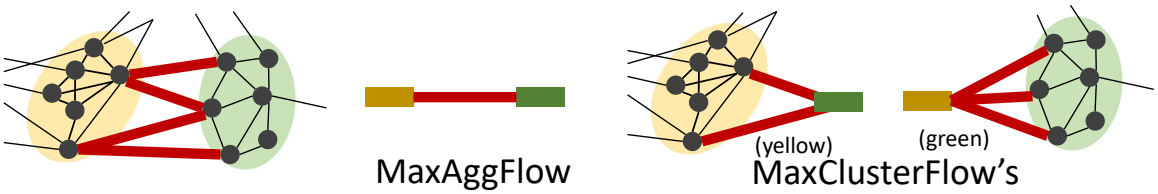


Figure A.1: Considering the crossing edges between the yellow and green clusters from Figure 3.3; MaxAggFlow has a single bundle; the yellow and green instances of MaxClusterFlow have one bundle for each incident node in their cluster.

equals the sum of the capacity of the underlying edges. Thus, the flow assigned to a bundle of edges by MaxAggFlow is no more than the total capacity of the edges in the bundle. Subsequently, MaxClusterFlow instances behave similarly; that is, the flow allocated for a bundle of edges is no more than the capacity of that bundle. For example, Figure A.1 shows the four edges between the yellow and green clusters in Figure 3.3 as well as the bundles considered by MaxAggFlow (in the middle) and the two instances of MaxClusterFlow corresponding to the yellow and green clusters on the right. The later steps in Figure 3.4 do not increase flow and so we conclude that capacity constraints are satisfiable for all non-local edges.

A.1.2 Proof that the heuristic in §3.3.2 leads to feasible flow allocations

Here, we prove Theorem 1. First, note that the heuristic in §3.3.2 which only restricts the edges between clusters and paths on the aggregate graph that can be used by some commodities does not affect the proof in §A.1.1; that is, edges still receive flow less than their capacity and demand constraints hold.

We now prove that the heuristic will satisfy flow conservation; that is, at any node in the network, for any commodity which neither originates nor ends at this node, the net flow is zero, i.e., incoming flow to the node equals the flow leaving that node.

It is easy to see that flow conservation holds for commodities whose source and target are in the same cluster because only the instance of MaxClusterFlow for that cluster assigns flow to such a commodity. Since MaxClusterFlow invokes FeasibleFlow in Equation 2.1, the flow is allocated along paths which start and end at the source and target of that commodity, respectively. Thus, every node that is neither the source or target will

have incoming flow equal to the outgoing flow. Note that flow conservation holds for this scenario (commodities whose source and target are in the same cluster) even without the heuristic in §3.3.2.

We now consider the remaining commodities, that is, whose source and target are in different clusters. In this case, it is possible for nodes to have edges to and from other clusters. Suppose by contradiction that some commodity k violates flow conservation at such a node u. The heuristic in §3.3.2 allocates flow for commodity k along only one path in the aggregated graph and on only one edge between connected clusters. If the cluster containing u is not on the chosen path or none of the chosen edges are incident on u, then the net flow allocated for k over all edges incident on u will be zero. Let e be that one chosen crossing edge incident on u which can receive non-zero flow for commodity k. Observe that all of the other commodities whose source and target are in the same clusters as k would also be allocated flow on the same path and edges as k. Thus, all the flow allocated for these commodities entering or leaving node u as the case may be would be on edge e. Two instances of MaxClusterFlow, one corresponding to the cluster that contains u and another corresponding to the other side of edge e, will assign possibly different flow values for this bundle of commodities on edge e. To conclude our proof, note that MinPathE2E takes the minimum flow assigned along all such crossing edges e on the chosen path through the aggregated graph and that SrcTargetMax further breaks open the bundle to assign feasible flow for each actual commodity contained in the bundle.

If more than one crossing edge or more than one path on the aggregate graph are used for a commodity, it is easy to see how the above proof will break. The two instances of MaxClusterFlow that correspond to the clusters on either side of a crossing edge will be forced by MinPathE2E to only agree on the total volume for the cluster bundle of commodities for all edges between the pair of clusters; that is, these instances may allocate different flow on different edges or allocate different flow to individual commodities in the bundle. Figure 3.6 shows simple examples of such disagreement.

A.1.3 Proof of optimality for algorithm in §3.3.1 given some sufficient conditions

Here, we prove Theorem 2. We already discussed in §3.3.3 the case where the number of clusters, η, is 1 or N, the number of nodes in the graph. To prove optimality for the other sufficient conditions, we posit a helper theorem.

Theorem 3. *Given a set of paths $\mathcal{P}$ that can be used by flows, there exists a clustering of nodes into clusters such that any flow allocated on a set of paths $\mathcal{P}$ can also be allocated by the method in Figure 3.4 over those clusters.*

Proof. The claim is trivially true by using N clusters, where each node is in a cluster by itself. We show that it is possible to use fewer clusters next. Let $\mathcal{S}$ be a set of nodes such that every path in $\mathcal{P}$ contains at most one contiguous sequence of the nodes in $\mathcal{S}$. For example, the set $\{u, v\}$ satisfies this property if every path in $\mathcal{P}$ has neither u nor v, just u but not v (no repetitions allowed), just v but not u, $u \rightarrow v$ (no repetitions of u or v anywhere else in the path) or $v \rightarrow u$. Coalescing each such set $\mathcal{S}$ into a cluster would allow the method in Figure 3.4 to allocate the same flow as MaxFlow using the paths in $\mathcal{P}$. □

If G_{agg} is a tree and there is at most one edge between any pair of clusters, any set of paths $\mathcal{P}$ on the actual graph would consist of contiguous segments that are contained within each cluster. Thus, per the above theorem, any flow allocated by MaxEdgeFlow (Equation A.4) can also be allocated by the method in Figure 3.4. The only difference then between the global optimization and the method in Figure 3.4 is that whereas the former is a single optimization call, the latter is a sequence of optimizations. Since demands are satisfiable, however, all of the steps in Figure 3.4 will allocate the entirety of demand and hence will allocate the maximum amount of flow.

Note, in particular, that for the sufficient conditions listed in Theorem 2 a single iteration of the steps in Figure 3.4 suffice.

In §A.7, we show some counter-examples where NCFlow can lead to sub-optimal allocations when any of these sufficient conditions do not hold.

A.2 Data-plane details for NCFlow

A.2.1 Actions at the NCFlow controller, after each allocation

The SDN controller for NCFlow computes total flow per commodity and some splitting ratios after each allocation.

Total Flow: The flow assigned to a commodity whose source and target are in different clusters is read off SrcTargetMax, i.e., $f_{4,k}$. For intra-cluster commodities, their flow is read off MaxClusterFlow, i.e., $f^x_{2,k}$ at the cluster x that contains the source and target of commodity k. These flow values are summed up over all the iterations used by NCFlow.

Splitting ratios at sources: At source s of cluster x, we have two cases depending on whether the target of the commodity is within the cluster x or in some other cluster y.

For the former case, let $\mathcal{P}_{st}$ be the path set to target t for commodity k; the splitting ratio for each path p in the set is $f^{x,p}_{2,k}$ summed up over all iterations, divided by the total flow assigned to commodity k above. Here, $f^{x,p}_{2,k}$ is the flow assigned to commodity k on path p by the MaxClusterFlow instance for cluster x.

For the latter case, let z_i be the next cluster on the one path that can receive flow in iteration i for all traffic going to targets in cluster y. The splitting ratio for path p in the path set $\bigcup_i \mathcal{P}_{sz_i}$ is the value of $\sum_{r \in K_{sy}} f^{x,p}_{2,r}$ summed up over all iterations where K_{sy} is the set of all commodities from source s to targets in cluster y divided by the total value for all such paths.

Uniquely, note that each source s has a splitting ratio per target t within the same cluster or per target cluster y.

We call a subset of nodes as ingresses if they have at least one edge to a node in another cluster that is chosen by the offline component of NCFlow in §3.3.4 as a crossing edge

Splitting ratios at ingresses are computed in a similar way to the splitting ratios at sources. At each ingress node w of cluster y for traffic from cluster x, there are two cases depending on whether the target is some node t in the same cluster as the ingress (y) or in some other cluster z.

For the former case, in iteration i, the splitting ratio for path p in the set $\mathcal{P}_{wt}$ is the value of $\sum_{r \in K_{xt}} f^{y,p}_{2,r}$ in iteration i divided by the total over all such paths. As above, K_{xt} is the

set of commodities from sources in cluster x to target t.

For the latter case, in iteration i, let z_i be the next cluster on the path to targets in z; the splitting ratio for path p in the set $\mathcal{P}_{wz_i}$ is the value of $\sum_{r \in K_{xz}} f_{2,r}^{y,p}$ divided by the total value over all such paths. As above, K_{xz} is the set of all commodities from sources in cluster x to targets in cluster z.

Note that an ingress node w has splitting ratios only for commodities whose chosen path at an iteration contains w's cluster (y) and whose chosen edge enters y at w.

A.2.2 Details on switch forwarding entries

Pathlets: NCFlow sets up label-switched paths (LSPs) between each pair of nodes in each cluster. Which paths to setup is pre-determined by the offline component in §3.3.4.

Splitting rules: A source s in cluster x has a splitting rule for each other node in the same cluster and for each other cluster. The splitting ratios are as computed in §A.2.1.

In each iteration, at each cluster, at most one ingress node is active per pair of other clusters. This is because the bundle of commodities for a given pair of clusters has at most one crossing edge entering a cluster.

The active ingress node at a cluster x for the bundle of commodities from cluster y to cluster z has one splitting rule when $z \neq x$ and one splitting rule per target in cluster x when $z = x$.

Packet content: The LSP (which pathlet to use) is encoded in the L2 header [122]. Additionally, NCFlow has the following tuple in each packet: (x, y, i, e) where x and y are the source and target cluster ids, i is the iteration number of the flow allocation that the packets have been assigned to and e is the edge to leave the current cluster on. The bits needed are $2 \ln \eta + \ln \mathcal{I} + \ln$ node degree.[1] We note that 16 bits of header space suffice for all the WAN topologies and experiments considered in this paper; that is $\eta \leq 64$ clusters, $\mathcal{I} \leq 8$ iterations and up to 2 edges to nodes in other clusters being used per egress node by NCFlow.

Data path actions:

[1]The edge id must suffice to distinguish at an egress node between the edges to a particular next cluster; so node degree is an overestimate.

- At source s in cluster x:
  - The host or middleware adds the cluster-ids x and y into the packet.
  - Source switch uses the appropriate splitting rule to pick a (p, i, e) tuple; the values e and i are placed in the packet and the L2 header gets the identifier for path p. To avoid reordering packets in the same TCP flow, traffic can be split using flow hashes or flowlets [78].
- Each cluster egress removes e from the packet header and forwards packets to the next-hop of the edge e.
- Each cluster ingress uses the appropriate splitting rule to pick a (p, e) tuple; the value e is put into the packet header and p determines the identifier in the L2 header.

A.3 Definitions of NoMoreFlow

In the flow vector computed by MaxClusterFlow at a cluster x, $\mathbf{f_2^x}$, we use the subscript k to denote a bundle that may include (1) transit commodities through cluster x (i.e., from all sources in some other cluster w to targets in some other cluster z), (2) leaving commodities (i.e., from a source in cluster x to all targets in some other cluster z) or (3) entering commodities (i.e., to a target in cluster x from all sources in some other cluster z). Furthermore, we use the subscript y_{out} to denote the flow allocated for the bundle k on paths to the virtual node that corresponds to the cluster y. Thus, $f^x_{2,k,y_{\text{out}}}$ is the flow allocated at cluster x for all commodities in the per-cluster bundle k on paths to the virtual node corresponding to a neighboring cluster y.

With this background, Equation A.1 ensures that the flows allocated in MinPathE2E for an inter-cluster bundle K in $\mathcal{D}_{\text{agg}}$ on all paths in $\mathcal{P}_{\text{agg}}$ that contain a cluster edge (x, y) is no more than the flow that is allocated at either cluster x or cluster y for their respective per-cluster bundles that are contained in K to and from each other respectively.

$$\mathsf{NoMoreAlongPaths}(\mathbf{f}, \mathbf{f_2}) \triangleq \forall K \in \mathcal{D}_{\text{agg}},\ \forall x, y \in \mathcal{V}_{\text{agg}}, x \neq y,$$
$$\sum_{p \in \mathcal{P}_{\text{agg}}, (x,y) \in p} f^p_K \leq \min\left(\sum_{k \in K} f^x_{2,k,y_{\text{out}}}, \sum_{k' \in K} f^y_{2,k',x_{\text{in}}}\right) \tag{A.1}$$

Equation A.2 is logically similar to Equation A.1 except that the constraints are specific to a cluster x and the constants and variables have been flipped; that is, here, the flows on the paths in the aggregate graph are given ($f^p_{1,K}$) and the flow on paths within the cluster are to be computed by MaxClusterFlow. In particular, note that $\sum_{p' \in \mathcal{P}_{x,*y}} f^{p'}_k \triangleq f^x_{2,k,y_{\text{out}}}$; that is, the flow assigned in MaxClusterFlow of cluster x on all paths leading to the virtual node corresponding to a neighbor cluster y is precisely the value on the right that is used above in Equation A.1.

$$\begin{aligned}
&\mathsf{NoMoreFlowThruCluster}(\mathbf{f}, \mathbf{f_1}, x) \triangleq \forall K \in \mathcal{D}_{\text{agg}},\ \forall y \in \mathcal{V}_{\text{agg}} : y \neq x, \\
&\sum_{p \in \mathcal{P}_{\text{agg}} : (y,x) \in p} f^p_{1,K} \geq \sum_{k \in K,\ p' \in \mathcal{P}_{x,y}} f^{p'}_k, \text{ and} \\
&\sum_{p \in \mathcal{P}_{\text{agg}} : (x,y) \in p} f^p_{1,K} \geq \sum_{k \in K,\ p' \in \mathcal{P}_{x,*y}} f^{p'}_k
\end{aligned} \tag{A.2}$$

A.4 Fault Model

When failures happen, prior works [27, 96] assume that the sources of the label switched paths (LSPs) will proportionally shift traffic. That is, a source that splits traffic in the ratio of $(0.3, 0.5, 0.2)$ between three paths will change to a splitting ratio of $(0.6, 0, 0.4)$ when the middle LSP fails. Doing so can cause congestion on either of the remaining LSPs.

The key idea in prior works [27,96] is to proactively allocate flow such that the maximal load on any link remains under capacity—FFC [96] protects against up to k simultaneous link failures, whereas TEAVAR [27] ensures that the flow at risk is below a given fraction (e.g., 99.9% of flow can be carried by the network on average over all possible failure scenarios).

The cost of such congestion protection is two-fold: *i)* proactive schemes substantially increase the solution runtime, and *ii)* they under-allocate flow, since capacity must be set aside to help with possible failures. Instead, NCFlow uses a *reactive* strategy, and recomputes a new flow allocation after the fault occurs. This enables NCFlow to carry more flow before the fault, and potentially carry more flow after recovery. Furthermore, since NCFlow uses fewer FIB entries for the same number of paths, it is naturally easier to spread flow

onto more paths with NCFlow. Thus, the key trade-off is slightly longer and more lossy episodes immediately after a fault when using NCFlow versus longer solver runtimes and flow under-allocation with proactive schemes [27, 96].

A.5 Benchmarking TEAVAR and TEAVAR*

A.5.1 Formulation for TEAVAR*

Here, we discuss our adaptation of TEAVAR to maximize total multi-commodity flow. The TEAVAR [27] paper considers a different objective – maximizing the *concurrent* multi-commodity flow (see Table 2.1). When all demands are satisfiable, both objectives allocate the same flow; however, when not enough capacity is available to meet the desired failure assurance, maximizing total flow leads to a strictly larger allocation. We describe TEAVAR* from first principles here.

In addition to the inputs of MaxFlow (see Equation 2.2), TEAVAR* has the following inputs:

- A value $\beta \in [0, 1]$; larger values of β correspond to greater fault assurance.
- A set of fault scenarios, $\mathcal{S}$; each scenario i has a probability of occurrence β_i and a set of failed edges $\mathcal{E}_i$.

In a fault scenario i, the edges in $\mathcal{E}_i$ will fail and so the flow allocated to paths that contain any edge in $\mathcal{E}_i$ will be *lost*. The number of possible fault scenarios is exponential in the number of edges in the network. Thus, to keep the optimization tractable, we consider only a subset of scenarios.

Let $\mathcal{L}(i)$ denote the total flow lost in fault scenario i. Per Proposition 8 in [121], minimizing the potential function, $\alpha + \frac{1}{1-\beta}\mathbb{E}[\mathcal{L}_i - \alpha]^+$, would minimize the conditional value at risk. Here, the expectation is over all possible fault scenarios. Since we only consider a subset of fault scenarios to keep optimization tractable, we minimize: $\alpha + \frac{1}{1-\beta}\left(\sum_{i\in\mathcal{S}} \beta_i[\mathcal{L}_i - \alpha]^+ + (1 - \sum_{i\in\mathcal{S}} \beta_i)(1-\alpha)\right)$. The last term accounts for the unconsidered scenarios for which we must assume the worst possible loss. Note that we can simplify this expression by dropping the constant $\frac{1-\sum_{i\in\mathcal{S}} \beta_i}{1-\beta}$.

$$
\begin{aligned}
&\mathsf{TEAVAR}^{\star}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P},\beta,\mathcal{S}) && && \text{(A.3)}\\
&\quad \triangleq \arg\min_{\mathbf{f}} \left(\alpha + \tfrac{1}{1-\beta}\left(\textstyle\sum_{i\in\mathcal{S}} \beta_i \mathsf{Excess}_i - (1-\textstyle\sum_{i\in\mathcal{S}}\beta_i)\alpha\right)\right) && &&\\
&\text{s.t. } \mathbf{f} \in \quad \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}), && && \text{(Eqn. 2.1)}\\
&\qquad \mathcal{L}_{i,k} \geq 0, && \forall i,k && \text{(loss is non-negative)}\\
&\qquad \mathcal{L}_{i,k} \geq d_k - \sum_{p\in\mathcal{P}_k} f_k^p \mathsf{Active}_{p,i}, && \forall i,k && \text{(loss)}\\
&\qquad \alpha \geq 0 && && \text{(loss } \textit{cutoff}\text{)}\\
&\qquad \mathsf{Excess}_i \geq 0 && \forall i && \text{(excess loss in scenario } i\text{)}\\
&\qquad \mathsf{Excess}_i \geq \sum_{k\in\mathcal{D}} \mathcal{L}_{i,k} - \alpha, && \forall i && \text{(excess loss)}
\end{aligned}
$$

The formulation for TEAVAR* is in Equation A.3. Recall that f_k^p is the flow assigned to demand k on path p. $\mathsf{Active}_{p,i}$ is an indicator denoting whether path p is active in fault scenario i. Thus, the allocation for demand k in scenario i will be $\sum_{p\in\mathcal{P}_k} f_k^p \mathsf{Active}_{p,i}$. When the allocation is below the required volume d_k, the demand will suffer loss; we use $\mathcal{L}_{i,k}$ to denote the flow loss for demand k in scenario i.

The flow allocation resulting from the above formulation cannot be promised to the demands; in particular, more flow will be assigned on some paths to account for possible failures on other paths. After solving the above LP, we compute the flow allocation for a demand k as follows: (1) sort the per-scenario losses $\mathcal{L}_{i,k}$ in ascending order; (2) starting at index 0, add up the probability of each scenario until the running sum is at least β—let i_β be the unique crossing index; (3) Set demand k's flow to be $d_k - \mathcal{L}_{i_\beta,k}$, the demand minus the loss at the crossing index.

Choosing the fault scenarios to use in TEAVAR*:

- Intuitively, achieving a greater amount of fault assurance requires considering more fault scenarios. Specifically, if the total probability of considered scenarios is below β, the above LP as well as the LP used by TEAVAR become unbounded. To see why, the coefficient of α in Eqn. A.3 is $\frac{\left(\sum_{i\in\mathcal{S}}\beta_i\right)-\beta}{1-\beta}$. If the probability of considered scenarios is less than β, this coefficient becomes negative, and the objective value reaches $-\infty$ by setting α to ∞.

- Intuitively, if the total probability of considered scenarios is just larger than β, the flow allocated to demands is very small. To see why, the smaller the value of $\sum_{i \in \mathcal{S}} \beta_i - \beta$, the smaller the positive coefficient of α in the objective of Eqn A.3. Thus, the solution of Eqn A.3 will have a large value of α and a very small amount of allocatable flow.
- In light of these two points, in our experiments, we choose all scenarios that are individually more likely to occur than a cutoff ρ and multiplicatively reduce ρ until the total probability of considered scenarios exceeds $1 - \frac{1-\beta}{2}$.

A.5.2 Comments on benchmarking TEAVAR

Observe that the number of scenarios affects the complexity of the TEAVAR* optimization; specifically, the number of equations and variables increases by $|\mathcal{S}| * |\mathcal{P}|$. The path set is at least as large as the node pairs, i.e., $|\mathcal{P}| > N^2$ where N is the number of nodes. The appropriate choice of fault scenarios to consider, as discussed above, depends on the size of the topology, the failure probability of edges, and the required assurance level β. Suppose one considers all 2-edge failure scenarios; then $|\mathcal{S}| \sim M^2$ where M is the number of edges. Hence, the increase in equations and variables exceeds N^2M^2. Note that MaxFlow is substantially simpler, having at most $O(N^2)$ variables and constraints (Equation 2.1).

On the topologies listed in Table 3.3, our implementation of TEAVAR* never ran to completion even after several days. We ran with $\beta = 0.99$ and link failure probability set to 0.004; both of these are the default values used in the open-source implementation.[2] The reason is that the optimization problem becomes intractably large. TEAVAR behaves similarly [27]. We conclude that probabilistic fault protection using this methodology is infeasible on large topologies and for non-trivial fault assurance levels such as when considering multiple link failures.

We also note that we are unable to simultaneously achieve the solution quality and the runtimes that are reported in TEAVAR [27] using their code. Specifically, achieving the assurance levels reported in their experiments requires many scenarios to be considered. The runtimes reported in [27] appear to have been measured when considering only single

[2] https://github.com/manyaghobadi/teavar

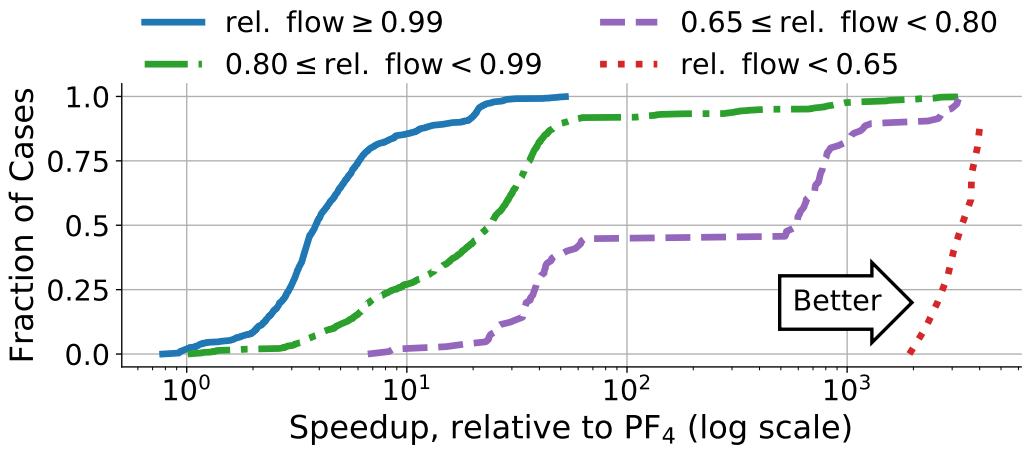


Figure A.2: Breaking down the NCFlow results from Figure 3.10b into four separate CDFs based on relative total flow.

link failures.

A.6 Additional Experiments

A.6.1 Breakdown of NCFlow's Performance

To further understand the performance of NCFlow, Figure A.2 breaks down the results in Figure 3.10 into four ranges based on total relative flow. We plot CDFs of the speedup ratio per range. The solid blue and green dashed line, which correspond to relative flow above 0.99 and in $[0.8, 0.99)$ respectively, account for 49% and 46% of all experiments. The figure shows that NCFlow achieves sizable speedups while allocating large amounts of flow.

Figure A.3 further breaks down the aggregate results from Figure 3.10 across various aspects of interest. In the two left-most columns, we break down the results by different settings of α, which illustrates how NCFlow performs on both under-subscribed ($\alpha = \{1, 8\}$) and over-subscribed ($\alpha = \{32, 64, 128\}$) traffic matrices. In the former case, NCFlow is typically able to fully satisfy the TM's requested demand, thereby matching the total flow allocated by the other methods. At the same time, NCFlow is strictly faster on all TMs, except for those belonging to smaller topologies (e.g., Uninett2010), which we discuss

Topology	Edge-Based	Räcke	KSP	NCFlow
Total # FIB Entries				
PrivateLarge	945,038,502	52,515,090	22,483,244	**1,694,027**
Kdl	427,524,786	76,794,001	30,199,751	**1,876,289**
PrivateSmall	7,684,182	1,232,866	625,282	**139,346**
Cogentco	7,567,952	2,054,323	915,207	**139,862**
UsCarrier	3,894,542	1,520,821	510,894	**82,301**
Colt	3,534,912	1,048,779	346,905	**67,307**
GtsCe	3,263,696	1,077,350	535,135	**101,368**
TataNld	3,006,720	1,062,629	540,088	**93,179**
DialtelecomCz	2,590,122	1,427,780	529,663	**83,128**
Ion	1,922,000	886,414	418,362	**71,614**
Deltacom	1,417,472	459,159	246,811	**53,948**
Interoute	1,306,910	483,960	249,979	**32,193**
Uninett2010	394,346	133,742	57,428	**21,185**
Maximum # FIB Entries				
PrivateLarge	962,361	828,397	313,850	**18,124**
Kdl	567,009	576,274	309,575	**16,926**
PrivateSmall	38,809	49,663	21,796	**3,639**
Cogentco	38,416	60,676	30,601	**3,144**
UsCarrier	24,649	41,897	17,822	**2,234**
Colt	23,104	47,077	17,344	**3,572**
GtsCe	21,904	36,070	15,477	**2,748**
TataNld	20,736	24,776	13,179	**2,104**
DialtelecomCz	18,769	34,014	11,084	**1,393**
Ion	15,376	25,261	12,954	**1,387**
Deltacom	12,544	25,135	13,029	**1,737**
Interoute	11,881	14,182	8,346	**710**
Uninett2010	5,329	8,891	3,626	**868**

Table A.1: Number of FIB entries for NCFlow vs. edge-based formulations (e.g., Fleischer-Edge), path-based formulations using Räcke Randomized Routing Trees (SMORE*), and path-based formulations using k-shortest paths (PF_4, Fleischer-Path, TEAVAR*) on every topology.

later on. As α increases, so, too, does NCFlow's runtime advantage; however, this does come at the cost of the total flow allocated. For example, when $\alpha = 32$, we see many instances where NCFlow is $> 100\times$ faster than PF_4, but allocates 75% of PF_4's total flow

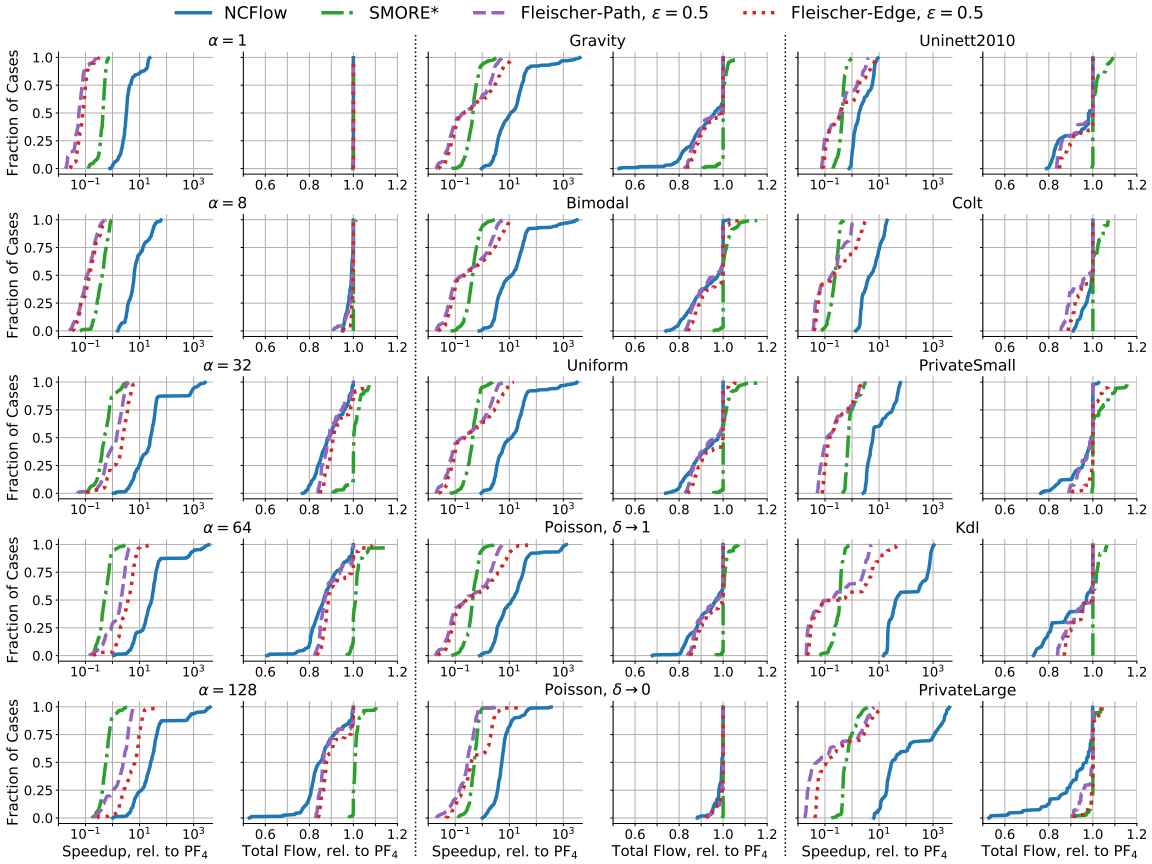


Figure A.3: A breakdown of the experimental results from Figure 3.10 along various dimensions of interest: scale factor, traffic model, and topology size. NCFlow excels on large topologies with TMs that have highly concentrated demands.

in the worst case. This effect becomes more evident for the largest settings of α: here, the speedups are $> 1000\times$, but more flow is sacrificed for some TMs. This behavior occurs perhaps because, as the traffic volume increases and the topology becomes more congested, paths that are not allowed by NCFlow's scheme become more critical for maximizing the total flow.

In the middle two columns, we break down the results by traffic model. NCFlow tends to perform best when demands are highly concentrated within clusters. In the bottom middle plot (Poisson, $\delta \to 0$), we see that NCFlow allocates $> 90\%$ of PF_4's total flow for almost every TM, while still achieving speedups $> 100\times$. Recall that as $\delta \to 0$ in the

Poisson traffic model, the traffic volume *between* clusters decreases, thus generating concentrated demands. In contrast, when $\delta \to 1$, demands are less concentrated, which leads to worse performance for NCFlow in terms of total flow, but not in terms of runtime.

Finally, in the two right-most columns, we break down the results by topology size. On Uninett2010, the smallest topology in our evaluation set, NCFlow's trade-off between total flow and runtime is not much better than the other baselines, particularly Fleischer-Edge.

As the topology size increases, NCFlow's advantage becomes more apparent. On Colt, NCFlow offers faster runtimes and sacrifices little flow, no more than 10% less than PF_4. On PrivateSmall and Kdl, NCFlow's speedup increases even more: $> 100\times$ faster than PF_4 on the majority of cases on Kdl. But flow is sacrificed, particularly for large values of α. However, NCFlow's trade-off is still favorable compared to other methods: for Kdl, we see multiple instances where NCFlow achieves 1,000$\times$ speedups at only a 20% reduction in flow. For PrivateLarge, we see both the biggest speedups and the smallest fraction of total flow relative to PF_4. As previously discussed, the outlier coincides with a highly oversubscribed TM ($\alpha = 128$). When we move to other regimes on PrivateLarge, NCFlow's performance improves: on 31 of the 400 TMs with $\alpha \in \{32, 64\}$, NCFlow is $> 1,000\times$ faster than PF_4 while achieving $> 80\%$ of PF_4's total flow.

In summary, we can see in this panel of CDF plots where NCFlow's strengths lie: on *i)* large topologies, and *ii)* TMs with moderate demands that are highly concentrated within the topology.

A.6.2 Alternate clustering methods

For each topology, we evaluate the three different clustering techniques mentioned in §3.3.4; on each topology we ask each technique to compute the number of clusters listed in Table 3.3. Figure A.4 shows CDFs of the ratio of total flow and latency speed-up of a clustering technique relative to that achieved by using FMPartitioning; thus values to the left of $x = 1$ indicate worse performance compared to FMPartitioning while those on the right indicate better performance. The figure shows that clusters discovered by FM partitioning almost always let NCFlow carry more flow (red lines); using either spectral clustering or leader election leads to a noticeably smaller allocation in about 20% and 40% of the cases.

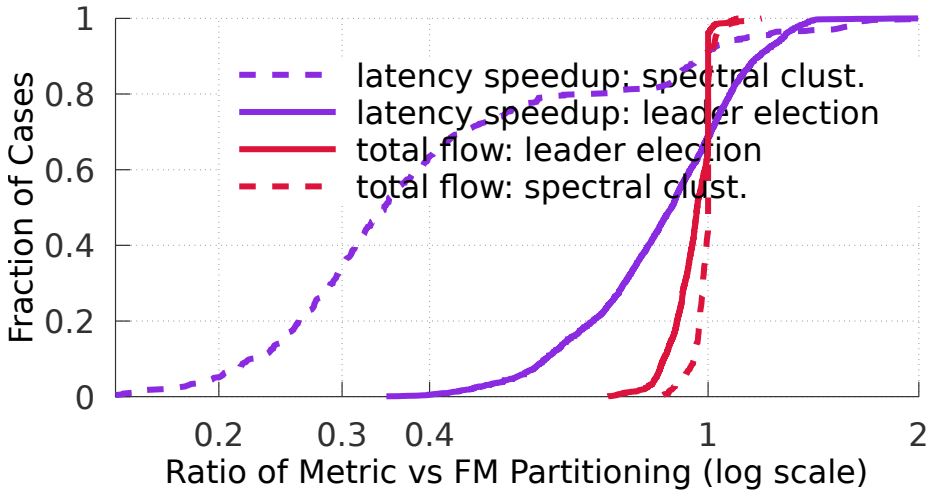


Figure A.4: Comparing the total flow allocated and the speedup in computing allocations when clusters are chosen using the three techniques mentioned in §3.3.4: FMPartitioning, Spectral Clustering and Leader Election. FMPartitioning, the default technique used in our evaluation, generally performs better, but not in all cases.

The figure shows a less clear-cut separation on latency speed-up; clusters discovered by leader election offer more speedup in over 30% of the experiments. Overall, we see that FMPartitioning performs better on average but not in all cases.

A.6.3 Effect on path latency

Figure A.5 shows a CDF of the *normalized path latency* for commodities[3] under different flow allocations. The figure on the top shows CDFs of the actual normalized path latency. Observe that these distributions are nearly identical. The figure on the bottom shows a CDF of the ratio of normalized latency; we see that roughly 70% of the commodities are carried by NCFlow on paths that are at most as long as the paths used by PF_4 (i.e., to the left of x=1). Most of the cases where NCFlow uses *relatively longer* paths are for commodities that have very small latency paths as illustrated by the top figure.

Note that path latency can be further explicitly controlled in NCFlow by determining

[3]The latency of the paths along which each commodity is routed weighted by the fraction of the commodity routed along each path. That is, if a commodity is divided equally between two paths, the normalized latency will be the average of the path latencies.

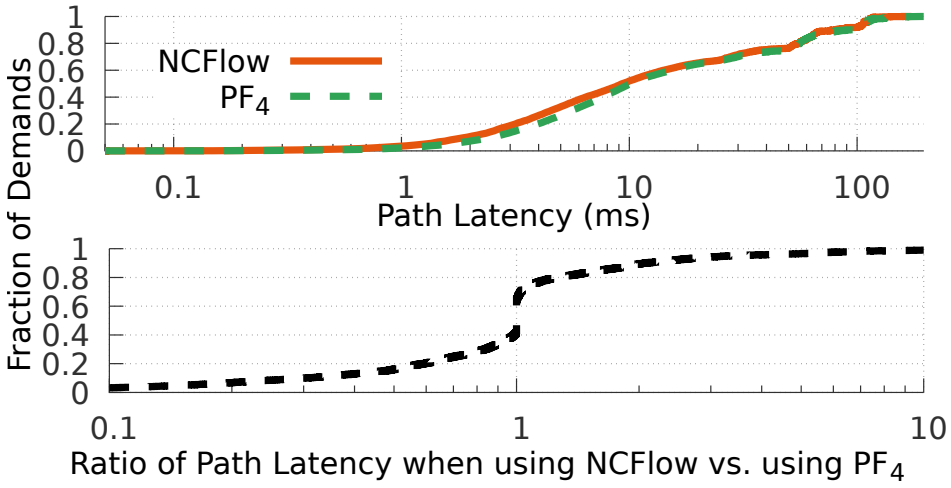


Figure A.5: Effect of NCFlow on path latency

which paths can be used or by weighting the objective to prefer shorter paths in the various steps of Figure 3.4.

A.6.4 Alternate path choices

With Figures A.6 to Figure A.10, we evaluate different numbers of paths between node pairs chosen with or without edge disjointness. PF_k refers to path formulation with k shortest paths chosen using edge disjointness and PF_{knd} indicates paths chosen without edge disjointness. Comparing these figures with Figure 3.10, we note that NCFlow's improvements over baselines hold across different path choices.

Note that Figure A.9 and Figure A.10 are missing some of the larger topologies listed in Table 3.3 for some of the baseline schemes because the baselines ran out of memory (we used a server with up to 3TB of memory) or raised some other exception.

A.7 Illustrative examples

Here, we show some illustrative examples where applying NCFlow using adversarially chosen clusters can lead to sub-optimal flow allocation.

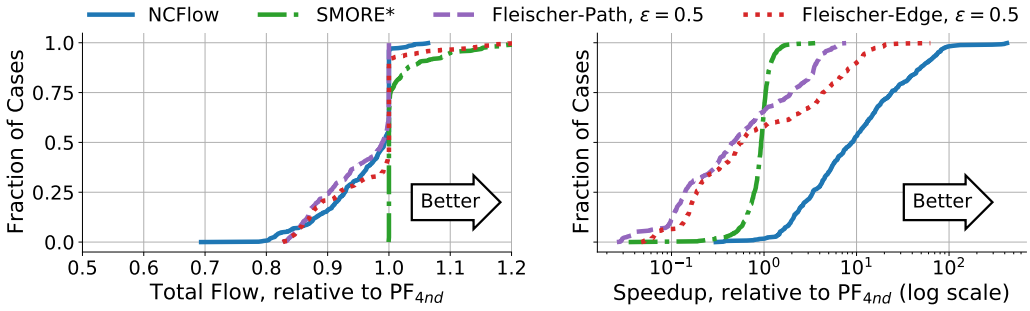


Figure A.6: Similarly to Figure 3.10 all schemes use up to $k = 4$ shortest paths between each pair of nodes except that the paths are chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 3.10.

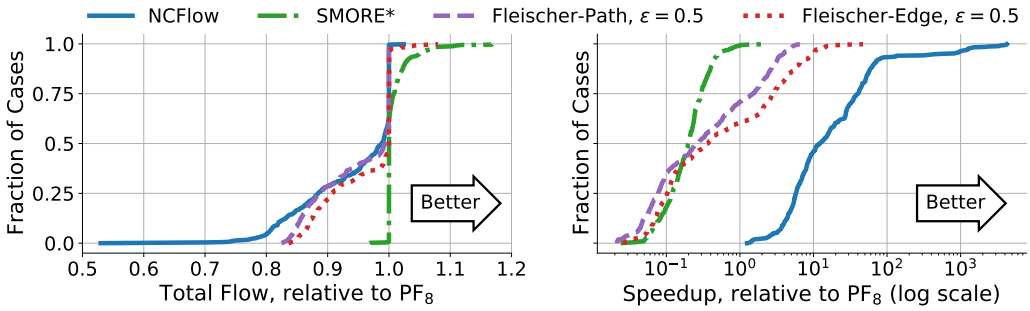


Figure A.7: Similar to Figure 3.10, except all schemes use up to $k = 8$ shortest paths between each pair of nodes; paths chosen *with* edge disjointness. The figure shows no qualitative difference relative to Figure 3.10.

Figure A.11 shows a case wherein NCFlow is sub-optimal because the aggregate graph (wherein nodes are clusters) is not a tree. The network topology and optimal allocations are shown in the graph on the left; assume each link has a unit capacity. With NCFlow, as shown in the figures on the right, MaxAggFlow can route the flow from s_1 to t_1 on either the top or the bottom path or divide between the two paths in some proportion; note that MaxAggFlow is not aware of demands that are local to a cluster (such as the flow from s_2 to t_2). Whenever MaxAggFlow assigns non-zero flow for the $s_1 \rightarrow t_1$ demand on the top path, NCFlow will be sub-optimal because then the other demand cannot be fully satisfied when MaxClusterFlow executes later on the yellow cluster. Any unsatisfied volume for $s_1 \rightarrow t_1$ can be routed on the bottom path in a later iteration but the flow for $s_2 \rightarrow t_2$ will not increase since the links that demand can use are fully utilized in the first iteration. The

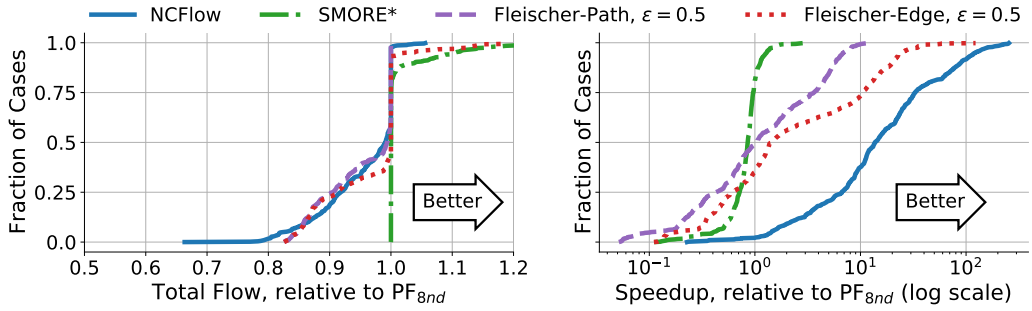


Figure A.8: Similar to Figure 3.10 except all schemes use up to $k = 8$ shortest paths between each pair of nodes; paths chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 3.10.

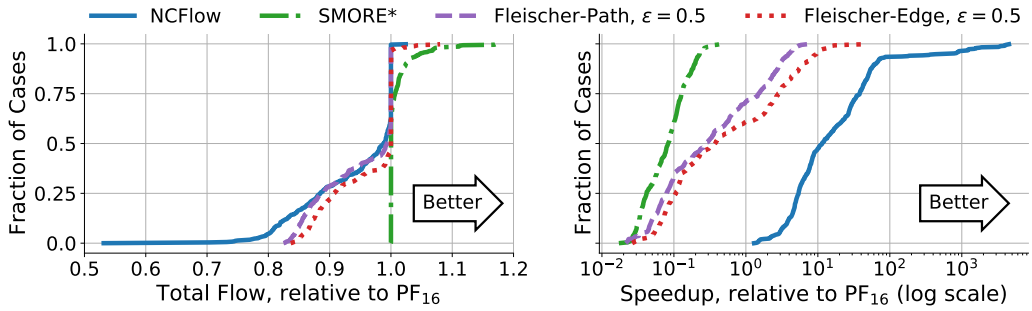


Figure A.9: Similar to Figure 3.10 except all schemes use up to $k = 16$ shortest paths between each pair of nodes; paths chosen *with* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 3.10.

root of the problem here is that MaxAggFlow allocates traffic over multiple paths without being aware of the demands within clusters.

Figure A.12 shows a case wherein NCFlow is sub-optimal when demands cannot be fully satisfied. As above, the topology and optimal allocations are shown on the left. Also, as above, the root of the issue here is that MaxAggFlow allocates the cross-cluster flow on the aggregate graph without being aware of the demands within clusters. As shown, subsequently, MaxClusterFlow will under-allocate flow for the local demands even though total flow would be larger if the local demands are fully satisfied.

Reordering the sub-problems, i.e., executing MaxClusterFlow before MaxAggFlow, may appear promising based on these examples but simple counter-examples exist even for such a reordered solution. The underlying cause of sub-optimality is not the order in

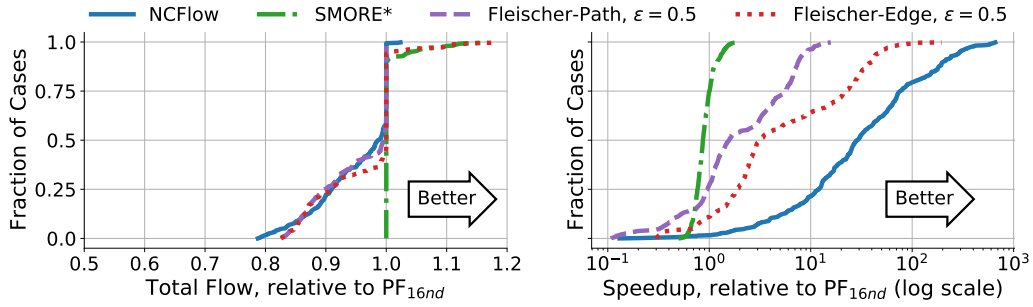
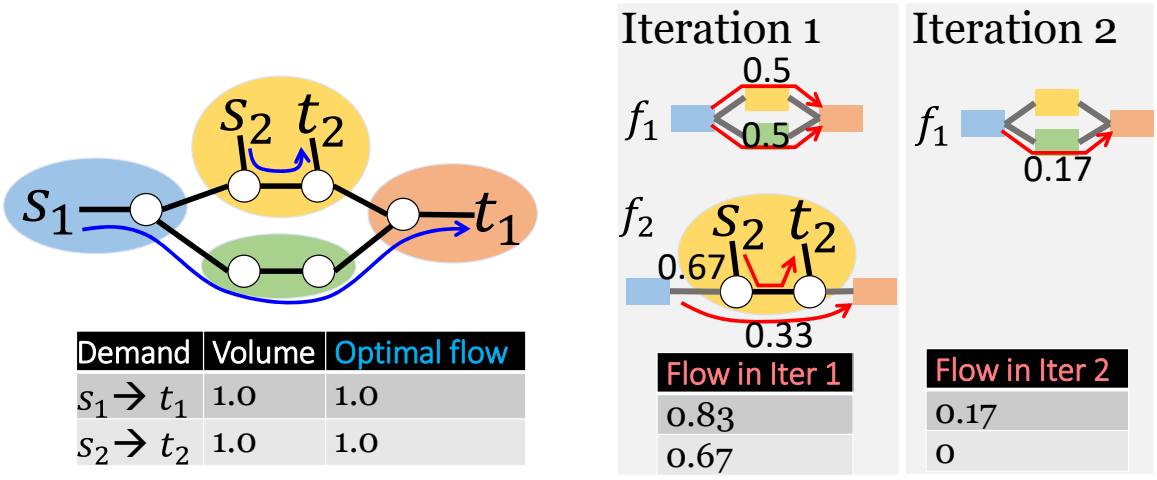


Figure A.10: Similar to Figure 3.10 except all schemes use up to $k = 16$ shortest paths between each pair of nodes; paths chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 3.10.



Demand	Volume	Optimal flow
$s_1 \rightarrow t_1$	1.0	1.0
$s_2 \rightarrow t_2$	1.0	1.0

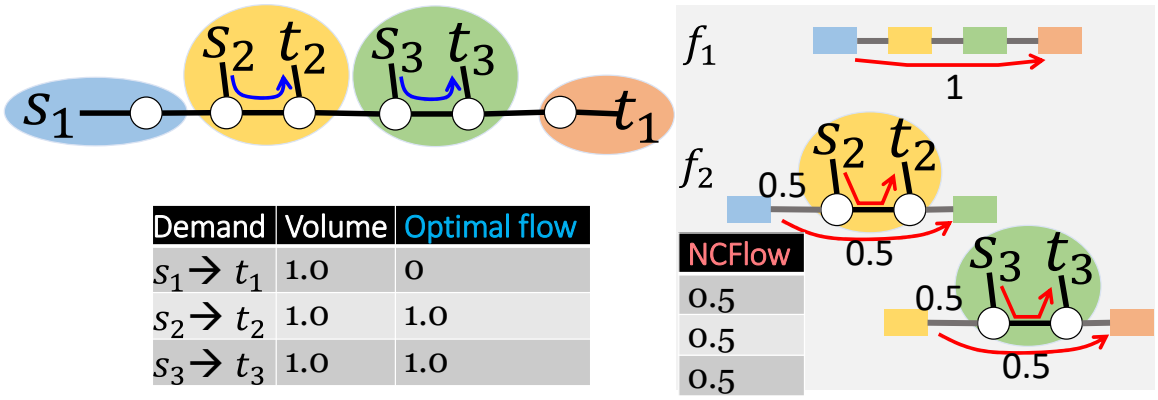
Flow in Iter 1
0.83
0.67

Flow in Iter 2
0.17
0

Figure A.11: Sub-optimality of NCFlow when the aggregate graph is not a tree.

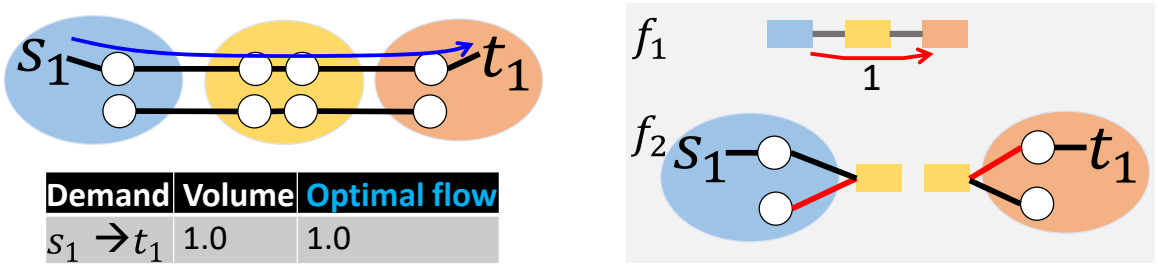
which the global and local solutions are computed but rather that the optimal flow allocation requires *jointly solving* these problems.

Figure A.13 shows a case wherein NCFlow can be sub-optimal when multiple edges connect clusters. As above, each unmarked link has unit capacity and the optimal allocations are shown in blue. Recall that NCFlow uses exactly one edge between each pair of clusters per iteration to avoid disagreements. There are two edges between each cluster but among the four possible crossing edge choices in an iteration, exactly one choice can carry non-trivial amount of flow (the top edge for each cluster pair). If that choice is somehow not picked, as shown marked in red on the right in Figure A.13, NCFlow will not satisfy



Demand	Volume	Optimal flow
$s_1 \rightarrow t_1$	1.0	0
$s_2 \rightarrow t_2$	1.0	1.0
$s_3 \rightarrow t_3$	1.0	1.0

Figure A.12: Sub-optimality of NCFlow when demands cannot be fully satisfied.



Demand	Volume	Optimal flow
$s_1 \rightarrow t_1$	1.0	1.0

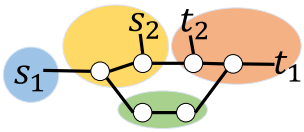
Figure A.13: Sub-optimality of NCFlow when there are multiple edges between pairs of clusters.

the demand. Simply increasing the number of iterations may not suffice either since the number of edge choices can be large, depending on the path lengths on the aggregate graph and on the number of edges between clusters.

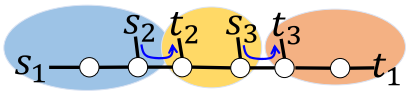
As noted previously, the above examples are in part due to poor cluster choices; Figure A.14 shows different cluster choices for these examples under which NCFlow will lead to optimal flow allocation.

A.8 Optimality gap

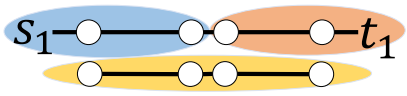
In this section, we analyze the gap in optimality between NCFlow and the best possible flow allocation for a hypothetical topology and traffic matrix.



(a) For the sub-optimality problem in Figure A.11, a different clustering choice that leads to optimal flow allocation with NCFlow.



(b) For the sub-optimality problem in Figure A.12, a different clustering choice that leads to optimal flow allocation with NCFlow.



(c) For the sub-optimality problem in Figure A.13, a different clustering choice that leads to optimal flow allocation with NCFlow.

Figure A.14: Alternate clustering choices that fix sub-optimality concerns and disagreements.

A.8.1 Optimal MaxEdgeFlow

The optimal flow allocation algorithm, in terms of carrying the maximum amount of flow possible on a network, is as shown in Equation A.4. We will call this the EF, short for Max Edge Flow. Some additional notation is in Table A.2. Observe that, in this formulation, any demand can be allocated on any edge (the variable f_{ke}) as long as flow conservation holds (the longer equation at the bottom). As noted in §3.2, this *edge formulation* of the problem carries the maximal amount of flow but has a high computation time and requires a large number of forwarding entries at switches (one rule per node pair at each node).

$$\text{MaxEdgeFlow}(\mathcal{V},\mathcal{E},\mathcal{D}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k \text{ s.t.} \tag{A.4}$$

$$\mathbf{f} = \{f_{ke} \mid \forall k \in \mathcal{D}, e \in \mathcal{E}\} \quad \text{and}$$

$$f_{ke} \geq 0 \qquad \forall e \in \mathcal{E}, k \in \mathcal{D} \quad \text{(non-negative flow)}$$

$$f_k \leq d_k, \qquad \forall k \in \mathcal{D} \quad \text{(below volume)}$$

$$\sum_{\forall k,e} f_{ke} \leq c_e, \qquad \forall e \in \mathcal{E} \quad \text{(below capacity)}$$

$$\sum_{e,u_e=u} f_{ke} - \sum_{e,v_e=u} f_{ke} = \begin{cases} f_k & \text{if } u = s_k \\ -f_k & \text{if } u = t_k \; \forall k \in \mathcal{D}, u \in \mathcal{V} \\ 0 & \text{o/w.} \end{cases} \quad \text{(flow cnsrvtn.)}$$

$u_e, v_e \in \mathcal{V}$	Edge $e \in \mathcal{E}$ goes from node u_e to node v_e
$m_u, \forall u \in \mathcal{V}$	m_u denotes the cluster containing node u. Note that $m_u \in \mathcal{V}_{\text{agg}}$ (i.e., the cluster is a node on the aggregate graph) and $u \in \mathcal{V}_{m_u}$ (i.e., the node u belongs in the restricted graph for the m_u'th cluster)
$\forall k \in \mathcal{D}, m_{s_k} \neq m_{t_k}, x \in \mathcal{V}_{\text{agg}}$	
$\mathsf{OutNodes}(x, k)$	The nodes in cluster x that can carry flow of demand k *out to* some other cluster, i.e., $\{u \mid m_u = x, \exists v \in \mathcal{V}, p \in \mathcal{P}_{\text{agg},K_{s_k t_k}}$ s. t. $m_v = y, (x, y) \in p, (u, v) \in \mathcal{E}\}$
$\mathsf{InNodes}(x, k)$	The nodes in cluster x that can carry flow of demand k *into* cluster x, i.e., $\{u \mid m_u = x, \exists v \in \mathcal{V}, p \in \mathcal{P}_{\text{agg},K_{s_k t_k}}$ s. t. $m_v = y, (y, x) \in p, (v, u) \in \mathcal{E}\}$

Table A.2: Additional notation for optimality gap; builds on top of notation from Table 2.1 and Table 3.1.

A.8.2 Edge flow with cluster constraints

Relative to the optimal MaxEdgeFlow, we first ask how much flow will be lost by using clusters. To compute this value, we add to MaxEdgeFlow the constraint shown in Equation A.5. Specifically, demands whose source and target are in the same cluster can only use edges within the cluster. However, as above, paths remain otherwise unconstrained.

$$f_{ke} = 0 \;\; \forall e, u_e \notin \mathcal{V}_x \text{ or } v_e \notin \mathcal{V}_x, \text{ if } m_{s_k} = m_{t_k} = x. \tag{A.5}$$

We will call this optimization problem EF with cluster constraints.

A.8.3 Path form with cluster and path constraints

Next, we ask how much flow will be lost when using the clusters as well as the given set of paths within and between clusters? Computing this value is somewhat more complex because we have to stitch together the flow carried on paths within each cluster with the flow on the edges between clusters while also ensuring that flow follow the chosen paths on

the aggregate graph (where clusters are nodes). For reference, we write this out in Equation A.6.

In more detail, this optimization problem, as shown in Equation A.6, has three classes of decision variables – f_K^p, f_k^p, f_{ke} – which respectively are the flow allocated to a bundled demand on a path on the aggregate graph, the flow allocated to a demand on a path within a cluster and the flow allocated to a demand on a crossing edge between clusters.

Equation A.7 computes the net flow for each demand k which for the case of a demand whose source and target are in the same cluster is the sum of flow carried on all intra-cluster paths. For demands whose source and target are in different clusters, the net flow is the flow from the demand's source to all of the nodes in the source's cluster that connect with other clusters as well as the flow to the demand's target from all of the nodes in the target's cluster that connect with other clusters.

For flow conservation, consider Equation A.9 which ensures that all of the flow leaving at a node u for a demand k on crossing edges to other clusters equals the flow that comes into the node u either from the source of the demand (if the source is within its cluster) or from all of the nodes in u's cluster that can receive flow for demand k from other clusters– $\mathsf{InNodes}(m_u, k)$. Equation A.10 considers the converse case for demands that leave at a node. Finally, Equation A.11 relates the total flow between a pair of clusters x, y on the crossing edges between these clusters with the flow along paths on the aggregate graph that contain the edge (x, y). We will call this optimization problem PF with cluster and path constraints.

$$\mathsf{MaxClusterPathFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k \qquad \text{s.t.} \quad \text{(A.6)}$$

$$\mathbf{f} = \big\{ f_K^p \mid \forall K \in \mathcal{D}_{\text{agg}}, p \in \mathcal{P}_{\text{agg}}, \quad \text{(flow on inter-cluster paths)}$$
$$f_k^p \mid \forall k \in \mathcal{D}, p \in \mathcal{P}, \quad \text{(flow on intra-cluster paths)}$$
$$f_{ke} \mid \forall k \in \mathcal{D}, e \in \mathcal{E}, m_{u_e} \neq m_{v_e} \text{(flow on edges between clusters)} \big\}$$

and

$$f_k = \begin{cases} \sum\limits_{p\in\mathcal{P}_{s_k,t_k}} f_k^p & \text{if } m_{s_k} = m_{t_k} \quad \text{(flow within a cluster)} \\ \sum\limits_{t\in\mathsf{OutNodes}_{(m_{s_k},k)}} \sum\limits_{p\in\mathcal{P}_{s_k,t}} f_k^p & \text{if } m_{s_k} \neq m_{t_k} \quad \text{(flow from source to outnodes)} \\ \sum\limits_{s\in\mathsf{InNodes}_{(m_{t_k},k)}} \sum\limits_{p\in\mathcal{P}_{s,t_k}} f_k^p & \text{if } m_{s_k} \neq m_{t_k} \quad \text{(flow to target from innodes)} \end{cases} \qquad \forall k \in \mathcal{D} \text{(net flow)}$$

(A.7)

$$f_k \leq d_k \text{(flow below volume)} \qquad \forall k \in \mathcal{D}$$

$$c_e \geq \begin{cases} \sum\limits_{k\in\mathcal{D}} \sum\limits_{p\in\mathcal{P},\ p\ni e} f_k^p & \text{if } m_{u_e} = m_{v_e} \quad \text{(intra-cluster edges; note: } k \text{ goes over all demands)} \\ \sum\limits_{k\in\mathcal{D}} f_{ke} & \text{otherwise} \quad \text{(inter-cluster edges)} \end{cases} \qquad \forall e \in \mathcal{E},$$

(A.8)

$$\sum_{e\in\mathcal{E}|u_e=u,\ m_{u_e}\neq m_{v_e}} f_{ke} = \begin{cases} \sum\limits_{p\in\mathcal{P}_{s_k u}} f_k^p & \text{if } m_u = m_{s_k} \quad \text{(at cluster } m_u\text{, flow from } s_k \text{ to } u\text{)} \\ \sum\limits_{v\in\mathsf{InNodes}(m_u,k)} \sum\limits_{p\in\mathcal{P}_{v,u}} f_k^p & \text{otherwise} \quad \text{(at cluster } m_u\text{, flow from all } \mathsf{InNodes} \text{ to } u\text{)} \end{cases} \qquad \forall u \in \mathcal{V}, k \in \mathcal{D}$$

(A.9)

$$\sum_{e\in\mathcal{E}|v_e=u,\ m_{u_e}\neq m_{v_e}} f_{ke} = \begin{cases} \sum\limits_{p\in\mathcal{P}_{u,t_k}} f_k^p & \text{if } m_u = m_{t_k} \quad \text{(at cluster } m_u\text{, flow from } u \textbf{ to } t_k\text{)} \\ \sum\limits_{v\in\mathsf{OutNodes}(m_u,k)} \sum\limits_{p\in\mathcal{P}_{u,v}} f_k^p & \text{otherwise} \quad \text{(at cluster } m_u\text{, flow from } u \text{ to all } \mathsf{OutNodes}\text{)} \end{cases} \qquad \forall u \in \mathcal{V}, k \in \mathcal{D}$$

(A.10)

$$\sum_{p\in\mathcal{P}_{\text{agg}}|(x,y)\in p} f_K^p = \sum_{e|m_{u_e}=x,\ m_{v_e}=y,\ k\in K} f_{ke} \qquad \forall K \in \mathcal{D}_{\text{agg}},\ x,\ y \in \mathcal{V}_{\text{agg}} \quad \text{(flow b/w clusters = flow on inter-cluster path)}$$

(A.11)

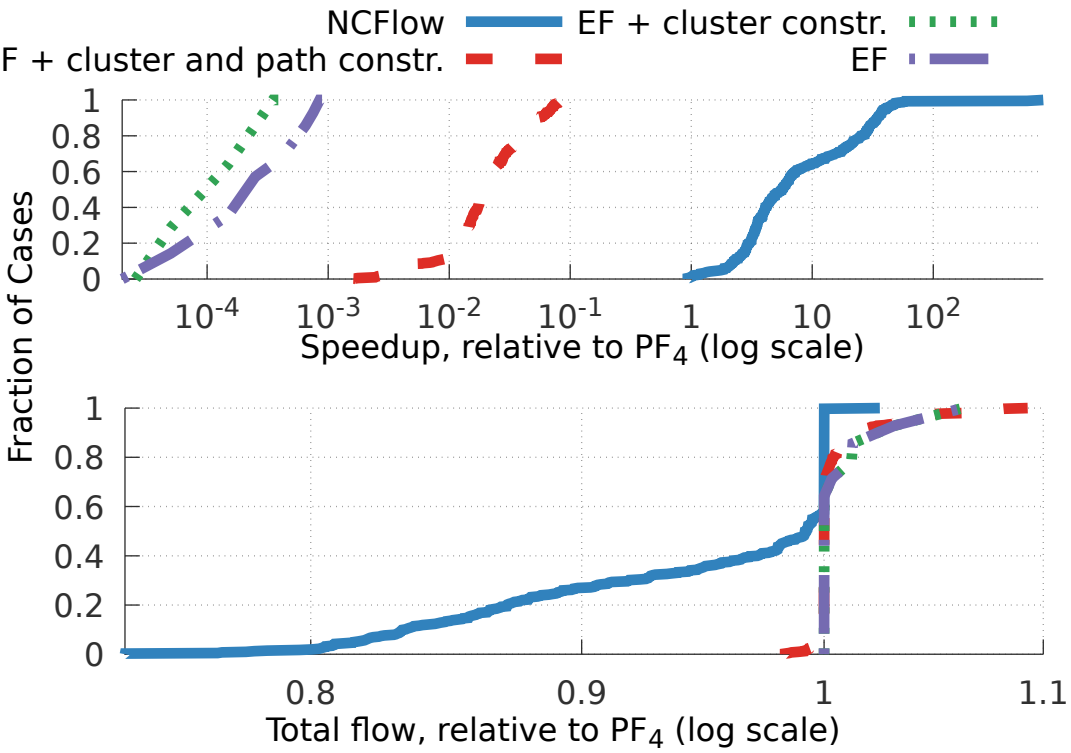


Figure A.15: Comparing flow allocated by NCFlow with the best possible flows

Note that the above constraints naturally lead to a reduction in forwarding table size as discussed in §3.3.5. However, it is not clear how much less flow these constraints allow for relative to the optimal EF. Moreover, since this optimization has more variables (and constraints) than PF_4 (see Equation 2.2), it can take longer to compute and may not be practically useful. We use this optimization problem to discern how much flow is lost by the constraints used in NCFlow (restricting to clusters and paths) relative to the flow that is lost due to the heuristic allocation process described in §3.3.

A.8.4 Experimental results

Our results are in Figure A.15; the baseline is PF_4 and the figures plot CDFs of total flow and latency speedup for many topologies and traffic demands. Note that using the edge formulation (purple dash-dots) often leads to substantially more flow being allocated compared to PF_4; however, as the figure on the top shows, edge formulation is a more

complex problem that takes longer to run (over $1000\times$ longer).

Adding the clustering constraint to edge formulation has an un-noticeable effect on the flow allocation (green dashes). Note that we use clusters computed using FMPartitioning for all topologies.

Constraining the path formulation using both the given clusters and the given paths (between clusters and within each cluster), as shown with the red dash line, allocates much more flow than PF_4 and not much less than is allocated in edge formulation. Thus, empirically, constraining flow allocation to traverse the chosen clusters and paths does not limit the flow that can be allocated. The figure also shows that computing the optimal flow given clusters and paths takes longer than PF_4 (roughly $10\times$ – $100\times$ longer). Thus, NCFlow offers a heuristic which finishes substantially faster than PF_4.

To sum up, our two main contributions are:

- constraining flow allocations to use specific clusters and paths which reduces the number of forwarding table entries needed without affecting the flow that can be allocated
- a heuristic that computes flow allocations quickly given this constraint but can under-allocate flow

We believe that future work can improve the heuristic to reduce the flow loss further.

Appendix B

POP

B.1 Proof of Bound on Random Partitioning for Simple Allocation Problem

In this section, we show a full derivation of Equation 6.2, which upper bounds the probability of a large gap between the optimal solution and solution returned by POP.

To quantify the gap between POP and optimal solutions, we need a sense of how big $q_{s,t}$ – the number of misplaced jobs of type s in sub-problem t – is in practice. In this section, we assume that the number of resources of each type are not necessarily equal; we define n_s as the number of resources of type s. We can compute a probabilistic upper bound on $q_{s,t}$ using a classical Chernoff bound, interpreting the random assignment of all type-s jobs (n_s of them) to sub-problems as Bernoulli trials where the probability that any given type-r job is placed in sub-problem l is $1/l$. Define $X_{s,t}$ to be the sum of all such trials, i.e., the number of type-s jobs in sub-problem t, with $E[X_{s,t}] = n_s/l$. Note that when $X_{s,t}$ exceeds the expected value, we get $X_{s,t} = n_s/l + q_{s,t}$. The Chernoff upper bound [100] can then be used to find the upper limit on the probability that the value of $X_{s,t}$ exceeds the expected

value by a fraction δ:

$$\begin{aligned}\Pr[X_{s,t} \geq (1+\delta)n_s/l] &= \Pr[q_{s,t} \geq \delta n_s/l] \\ &\leq \exp\left(\frac{-\delta^2 n_s}{(2+\delta)l}\right) \end{aligned} \tag{B.1}$$

In the rest of this text, to simplify notation, we will refer to the RHS of Equation B.1 as $C(\delta, n_s, l)$. For a simple problem with $r = 2$ and $l = 2$, if we have $n = m = 10^5$ jobs and resources split equally across resource types, the probability of exceeding the expected amount of type A jobs in a given sub-problem by 1% is 0.2877, by 2% is 0.00694, and by 3% is 0.0000145.

This bound can be extended to misplaced jobs across all resource types and sub-problems using the union bound, i.e., $\Pr(Z_1 \vee Z_2) \leq \Pr(Z_1) + \Pr(Z_2)$. This can be used to compute an upper limit on the probability that any resource type exceeds its expectation by a fraction $(1+\delta)$ on any sub-problem. Define $Y_{s,t}$ to be the event that type-r jobs in sub-problem l are in excess of the expected amount by a factor of $(1+\delta)$, i.e., $X_{s,t} \geq (1+\delta)n_s/l$. Then, we see that the following holds:

$$\Pr[Y_{s,1} \vee ... \vee Y_{s,l}] \leq \sum_{t=1}^{l} \Pr[Y_t] \leq \sum_{t=1}^{l} C(\delta, n_s, l) \tag{B.2}$$

We can extend this to all resource types similarly. Let Z_r be the probability that type-s jobs in any sub-problem l exceeds $(1+\delta)n_s/l$. Using the union bound again, we can extend Equation B.2 to compute the upper limit on the probability that the total number of misplaced jobs exceeds δn.

$$\begin{aligned}&\Pr\left[\sum_{s=1}^{r}\sum_{t=1}^{l} q_{s,t} \geq \delta n\right] \leq \Pr[Z_1 \vee ... \vee Z_R] \\ &\leq \sum_{s=1}^{r} \Pr[Z_j] \leq \sum_{s=1}^{r}\sum_{t=1}^{l} C(\delta, n_s, l)\end{aligned} \tag{B.3}$$

We can now combine this with Equation 6.1 to bound the performance of a randomized POP solution for the simplified allocation problem discussed in §6.7.1. We define Γ^* to be

an optimal allocation, Γ^{POP} to be the allocation returned by the POP procedure, and $U() : \Gamma \rightarrow u$ to be a function that computes the utility of an allocation Γ. Using Equations 6.1 and B.3, the probability that a random job partition will result in a utility that is greater than $\delta u_{\text{maxgap}} n$ from optimal is:

$$\begin{aligned}
&\Pr[U(\Gamma^*) - U(\Gamma^{\text{POP}}) \geq \delta u_{\text{maxgap}} n] \\
&\qquad \leq \Pr\left[\sum_{s=1}^{r}\sum_{t=1}^{l} q_{s,t} u_{\text{maxgap}} \geq \delta u_{\text{maxgap}} n\right] \\
&\qquad \leq \sum_{s=1}^{r}\sum_{t=1}^{l} C(\delta, n_s, l)
\end{aligned}$$

Bibliography

[1] Contraction Hierarchies Path Finding Algorithm. https://www.mjt.me.uk/posts/contraction-hierarchies/.

[2] Google OR-Tools. https://developers.google.com/optimization.

[3] Internet Live Stats. https://www.internetlivestats.com/one-second/#traffic-band.

[4] Kubernetes. https://github.com/kubernetes/kubernetes.

[5] Market Trends: SD-WAN and NFV for Enterprise Network Services. https://www.gartner.com/en/documents/3980319.

[6] Microsoft global network - Azure — Microsoft Docs. https://docs.microsoft.com/en-us/azure/networking/microsoft-global-network.

[7] MOSEK Optimization Suite. https://www.mosek.com/.

[8] OpenShift. https.//openshift.com.

[9] Polynomial-Time Approximation Scheme. https://en.wikipedia.org/wiki/Polynomial-time_approximation_scheme.

[10] What is BGP? — BGP routing explained — Cloudflare. https://www.cloudflare.com/learning/security/glossary/what-is-bgp/.

[11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms*, 2011.

[12] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting Wide-area Network Topologies to Solve Flow Problems Quickly. In *NSDI*, 2021.

[13] Akshay Agrawal, Stephen Boyd, Deepak Narayanan, Fiodar Kazhamiaka, and Matei Zaharia. Allocation of Fungible Resources via a Fast, Scalable Price Discovery Method. *arXiv preprint arXiv:2104.00282*, 2021.

[14] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A Rewriting System for Convex Optimization Problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[15] Ravindra Ahuja, Thomas Magnanti, and James Orlin. *Network Flows. Theory, Algorithms, and Applications*. Prentice Hall.

[16] Muthukaruppan Annamalai et al. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *OSDI*, 2018.

[17] D. Applegate, L. Breslau, and E. Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *SIGMETRICS*, 2004.

[18] David Applegate and Edith Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[19] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels, 2001. IETF RFC 3209.

[20] Daniel Awduche, Angela Chiu, Anwar Elwalid, Indra Widjaja, and XiPeng Xiao. Overview and principles of internet traffic engineering. Technical report, 2002.

[21] Daniel Awduche, Joe Malcolm, Johnson Agogbua, Mike O'Dell, and Jim McManus. Requirements for traffic engineering over MPLS, 1999.

[22] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. Capacity Planning for the Google Backbone Network. 2015.

[23] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *CoRR*, 2015.

[24] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to Linear Optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[25] Daniel Bienstock. *Potential function methods for approximately solving linear programming problems: theory and practice*, volume 53. Springer Science & Business Media, 2002.

[26] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks, 2008.

[27] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: Striking the Right Utilization-Availability Balance in WAN Traffic Engineering. In *SIGCOMM*, pages 29–43. 2019.

[28] Stephen Boyd, Neal Parikh, and Eric Chu. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine Learning*, pages 1–122, 2011.

[29] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[30] Stephen Boyd, Lin Xiao, Almir Mutapcic, and Jacob Mattingley. Notes on Decomposition Methods. *Notes for EE364B, Stanford University*, 635:1–36, 2007.

[31] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On Modularity Clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007.

[32] Matteo Brucato, Juan Felipe Beltran, Azza Abouzied, and Alexandra Meliou. Scalable Package Queries in Relational Database Systems. *Proc. VLDB Endow.*, 9(7):576–587, March 2016.

[33] P. Brucker. On the complexity of clustering problems. In *Optimizing and Operations Research*, Berlin, West Germany, 1977. Springer-Verlag.

[34] Matthew Caesar. BGP routing policies in ISP networks. *Business*.

[35] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM Computer Communication Review*, 37(4):1, October 2007.

[36] Martín Casado, Nick McKeown, and Scott Shenker. From Ethane to SDN and Beyond. *ACM SIGCOMM Computer Communication Review*, 49(5):92–95, 2019.

[37] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *NSDI*, 2017.

[38] P. Chardaire and A. Lisser. Simplex and Interior Point Specialized Algorithms for Solving Nonoriented Multicommodity Flow Problems. *Operations Research*, 2002.

[39] Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic Engineering with Equal-Cost-Multipath: An Algorithmic Perspective. *IEEE/ACM Transactions on Networking*, 25(2):779–792, 2016.

[40] Yi-Ching Chiu, Brandon Schlinker, Abhishek Balaji Radhakrishnan, Ethan Katz-Bassett, and Ramesh Govindan. Are We One Hop Away from a Better Internet? In *IMC*, pages 523–529, 2015.

[41] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *OSDI*, pages 430–446, 2019.

[42] A. Clauset, M.E.J. Newman, and C. Moore. Finding Community Structure in Very Large Networks. *Phys. Rev.*, 2004.

[43] Aaron Clauset. Fast Modularity Community Structure Inference Algorithm. https://www.cs.unm.edu/~aaron/research/fastmodularity.htm, 2019.

[44] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving Linear Programs in the Current Matrix Multiplication Time. *Journal of the ACM (JACM)*, 68(1):1–39, 2021.

[45] IBM ILOG Cplex. V12.1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[46] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD*, pages 313–324, 2011.

[47] George B Dantzig and Philip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, 1960.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[49] Nikhil R Devanur, Kamal Jain, Balasubramanian Sivan, and Christopher A Wilkens. Near Optimal Online Algorithms and Fast Approximation Algorithms for Resource Allocation Problems. In *EC*, pages 29–38, 2011.

[50] Steven Diamond and Stephen Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[51] Nick Feamster and Florham Park. Guidelines for Interdomain Traffic Engineering. *Computer Communications*, 33(5):19–30, 2003.

[52] Nick Feamster and Jennifer Rexford. A Model of BGP Routing for Network Engineering. *Engineering*, 2004.

[53] Lisa K Fleischer. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000.

[54] Ken Florance. How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience. https://bit.ly/2RYYrEM, 2016.

[55] B. Fortz and M. Thorup. Robust Optimization of OSPF/IS-IS Weights. In *INOC*, 2003.

[56] B. Fortz and Mikkel Thorup. Internet Traffic Engineering by Optimizing OSPF Weights in a Changing World. In *INFOCOM*, 2000.

[57] B. Fortz and Mikkel Thorup. Optimizing OSPF/IS-IS Weights in a Changing World. In *IEEE JSAC*, 2002.

[58] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. Traffic Engineering with Traditional IP Routing Protocols. *IEEE Communications Magazine*, 40(10):118–124, 2002.

[59] Naveen Garg and Jochen Könemann. Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. *SIAM J. Comput.*, 37(2):630–652, May 2007.

[60] Jamie Gaudette and Tim Stuch. Open Undersea Cable Systems for Cloud Scale Operation. In *Optical Fiber Communication Conference (OFC'17)*. OSA, March 2017.

[61] A. M. Geoffrion and G. W. Graves. Multicommodity Distribution System Design by Benders Decomposition. *Management Science*, 1974.

[62] Arthur M Geoffrion. Generalized Bender's Decomposition. *Journal of optimization theory and applications*, 10(4):237–260, 1972.

[63] Kostas Giotis, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. Combining OpenFlow and sFlow for an Effective

and Scalable Anomaly Detection and Mitigation Mechanism on SDN Environments. *Computer Networks*, 62:122–136, 2014.

[64] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet Routing. In *SIGCOMM*, 2009.

[65] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[66] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a Scalable and Flexible Data Center Network. *ACM SIGCOMM Computer Communication Review*, 39(4), 2009.

[67] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, pages 485–500, 2019.

[68] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi Optimizer Reference Manual, version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[69] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[70] Jeff Hartline and Alexa Sharp. Hierarchical Flow. Technical Report 2004-09-29, Cornell University, 2004.

[71] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[72] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4

and after: managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *SIGCOMM*, 2018.

[73] Christian Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. Technical report, 2000.

[74] Yuanfang Hu, Yi Zhu, Hongyu Chen, Ronald L. Graham, and Chung-Kuan Cheng. Communication latency aware low power NoC synthesis. In *DAC*, 2006.

[75] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, pages 261–276, 2009.

[76] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[77] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic Pricing and Traffic Engineering for Timely Inter-Datacenter Transfers. In *SIGCOMM*, 2016.

[78] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic Load Balancing Without Packet Reordering. In *CCR*, 2006.

[79] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for Wide Area Networks. In *SIGCOMM*, 2014.

[80] George Karakostas. Faster Approximation Schemes for Fractional Multicommodity Flow Problems. *ACM Transactions on Algorithms (TALG)*, 4(1):1–17, 2008.

[81] Yousef Khalidi. How Microsoft Builds Its Fast and Reliable Global Network. https://azure.microsoft.com/en-us/blog/how-microsoft-builds-its-fast-and-reliable-global-network, 2017.

[82] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[83] Robert Krauthgamer, Joseph (Seffi) Naor, and Roy Schwartz. Partitioning Graphs into Balanced Components. In *SODA*, 2009.

[84] Krishnaswamy, Umesh and Singh, Rachee and Bjørner, Nikolaj and Raj, Himanshu. Decentralized cloud wide-area network traffic engineering with {BLASTSHIELD}. In *NSDI*, pages 325–338, 2022.

[85] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*, pages 1–14, 2015.

[86] Praveen Kumar et al. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *NSDI*, 2018.

[87] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. YATES: Rapid Prototyping for Traffic Engineering Systems. In *SOSR*, 2018.

[88] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. *ACM SIGCOMM Computer Communication Review*, 40(4):75–86, 2010.

[89] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-Datacenter Bulk Transfers with NetStitcher. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 74–85. ACM, 2011.

[90] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing. In *SOSR*, 2018.

[91] Yin Tat Lee and Aaron Sidford. Efficient Inverse Maintenance and Faster Algorithms for Linear Programming. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 230–249. IEEE, 2015.

[92] Tony Li and Henk Smit. IS-IS Extensions for Traffic Engineering. Technical report, 2008.

[93] Xiaoqian Li and Kwan L Yeung. Traffic Engineering in Segment Routing Networks Using MILP. *IEEE Transactions on Network and Service Management*, 17(3):1941–1953, 2020.

[94] Chansook Lim, S. Bohacek, Joao Hespanha, and Katia Obraczka. Hierarchical Max-Flow Routing. In *Globecom*, 2005.

[95] Igor Litvinchev and Vladimir Tsurkov. *Aggregation in Large-Scale Optimization*, volume 83. Springer Science & Business Media, 2013.

[96] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[97] Richard McBride. Progress Made in Solving the Multicommodity Flow Problem. *SIAM Journal on Optimization*, 1998.

[98] M Meyer, J Vasseur, D Maddux, C Villamizar, and A Birjandi. MPLS Traffic Engineering Soft Preemption. Technical report, RFC 5712, 2010.

[99] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[100] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge university press, 2017.

[101] Srinivas Narayana, Joe Jiang, Jennifer Rexford, and Mung Chiang. Distributed Wide-Area Traffic Management for Cloud Services. In *SIGMETRICS*, 2012.

[102] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *SOSP*, pages 521–537, 2021.

[103] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[104] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. GAP:Generalizable Approximate Graph Partitioning Framework. 2019.

[105] NetworkX. Edge Disjoint Paths. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.connectivity.disjoint_paths.edge_disjoint_paths.html.

[106] Andrew Newell, Dimitrios Skarlatos, et al. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *28th ACM Symposium on Operating Systems Principles*, 2021.

[107] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2002.

[108] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. SCS: Splitting Conic Solver, version 2.1.3. https://github.com/cvxgrp/scs.

[109] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016.

[110] Brendan O'Donoghue, Giorgos Stathopoulos, and Stephen Boyd. A Splitting Method for Optimal Control. *IEEE Transactions on Control Systems Technology*, 21(6):2432–2442, 2013.

[111] Murat Oguz, Tolga Bektas, and Julia A. Bennell. Multicommodity flows and Benders decomposition for restricted continuous location problems. *European Journal of Operational Research*, 2017.

[112] David Oran. Osi is-is intra-domain routing protocol. Technical report, 1990.

[113] James Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Programming*, 1997.

[114] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[115] Abhinav Pathak, Ming Zhang, Y Charlie Hu, Ratul Mahajan, and Dave Maltz. Latency Inflation with MPLS-based Traffic Engineering. In *IMC*, pages 463–472, 2011.

[116] Peter Phaal, Sonia Panchen, and Neil McKee. Rfc3176: Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks, 2001.

[117] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low Latency Geo-distributed Data Analytics. In *SIGCOMM*, 2015.

[118] Harald Räcke. Minimizing Congestion in General Networks. In *FOCS*, pages 43–52. IEEE, 2002.

[119] Ragheb Rahmaniani, Teodor Gabriel Crainic, Michel Gendreau, and Walter Rei. The Benders Decomposition Algorithm: A Literature Review. *European Journal of Operational Research*, 259(3):801–817, 2017.

[120] Sajjad Rizvi, Xi Li, Bernard Wong, Fiodar Kazhamiaka, and Benjamin Cassell. Mayflower: Improving Distributed Filesystem Performance through SDN / Filesystem Co-Design. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 384–394. IEEE, 2016.

[121] R Tyrrell Rockafellar and Stanislav Uryasev. Conditional Value-at-Risk for General Loss Distributions. *Journal of banking & finance*, 26(7):1443–1471, 2002.

[122] E. Rosen, A. Viswanathan, and R. Callon. Multi-protocol Label Switching Architecture. RFC 3031.

[123] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *IMW*, 2002.

[124] Raja R Sambasivan, David Tran-Lam, Aditya Akella, and Peter Steenkiste. Bootstrapping Evolvability for Inter-domain Routing with D-BGP. In *SIGCOMM*, pages 474–487, 2017.

[125] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *SIGCOMM*, 2017.

[126] Submarine Cable Map. http://www.submarinecablemap.com.

[127] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *VLDB*, 7(12):1035–1046, 2014.

[128] Shaun Ray. AWS Global Accelerator for Availability and Performance. https://aws.amazon.com/global-accelerator/, 2018.

[129] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, pages 225–238, 2012.

[130] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *OSDI*, pages 827–844, 2020.

[131] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *VLDB*, 8(3):245–256, 2014.

[132] Dave Thaler and C Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. Technical report, 2000.

[133] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The Next Generation. In *EuroSys*, pages 1–14, 2020.

[134] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *EuroSys*, pages 1–16, 2016.

[135] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to Route. In *HotNets*, 2017.

[136] Robert J Vanderbei et al. *Linear Programming*, volume 3. Springer, 2015.

[137] Santosh Vempala, Ravi Kannan, and Adrian Vetta. On Clusterings Good, Bad and Spectral. In *FOCS*, 2000.

[138] Curtis Villamizar. Ospf Optimized Multipath (OSPF-OMP), 1999. Internet Draft.

[139] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *ACM SIGCOMM*, 2006.

[140] I-Lin Wang. Multicommodity Network Flows: A Survey, Part I: Applications and Formulations. *Internal Journal of Operations Research*, 2018.

[141] Mathieu Xhonneux, Fabien Duchêne, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with IPv6 Segment Routing. In *CoNext*, 2018.

[142] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[143] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew J. Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Muhammad Mukarram Bin Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *SIGCOMM*, 2017.

[144] Jin Yen and NetworkX. K-Shortest Paths. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.simple_paths.shortest_simple_paths.html.

[145] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[146] Rui Zhang-Shen and Nick McKeown. Designing a Predictable Internet Backbone with Valiant Load-Balancing. In *International Workshop on Quality of Service*, pages 178–192. Springer, 2005.

[147] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. ARROW: Restoration-Aware Traffic Engineering. In *SIGCOMM*, pages 560–579, 2021.