Real Time Alpha-fairness Based Traffic Engineering

Bill McCormick
Huawei Technologies Canada
Kanata Ontario, Canada
bill.mccormick@huawei.com

Frank Kelly
University of Cambridge
Cambridge, UK
f.p.kelly@statslab.cam.ac.uk

Patrice Plante
Huawei Technologies Canada
Kanata, Ontario, Canada
patrice.plante@huawei.com

Paul Gunning
BT Research & Innovation
Adastral Park, Ipswich, UK
paul.gunning@bt.com

Peter Ashwood-Smith
Huawei Technologies Canada
Kanata, Ontario, Canada
peter.ashwoodsmith@huawei.com

ABSTRACT

SDN traffic engineering is used to assign bandwidth to flows. Classic traffic engineering algorithms are well understood however implementations of these algorithms typically take seconds or even minutes to execute. These long execution times force traffic engineering to be used as an off-line tool. We demonstrate a traffic engineering algorithm equivalent to these classic algorithms that executes in millisecond times, allowing traffic engineering to be used as an on-line tool – much as shortest path computations are used in today's routers.

Categories and Subject Descriptors

G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Parallel Algorithms*

Keywords

Software defined networking; Traffic engineering

1. INTRODUCTION

Elastic flows - characteristic of TCP - will attempt to consume all the bandwidth available in the network. When conducting traffic engineering, two key concepts are the total network throughput, and providing fairness between the flows on the network.

We balance network throughput and fairness using the concept of α-fairness introduced by Mo and Walrand in [2] where α is a parameter in the range $[0, \infty]$. Small values of α favour network throughput, and large values of α favour fairness. As $\alpha \to \infty$, the flow assignment approaches the well known max-min fairness.

2. ALGORITHM DESCRIPTION

In [4] the author develops a stability proof for a primal-dual convex algorithm intended for congestion management of a future TCP implementation. Our work is developed from [4] by centralizing this distributed algorithm.

Model the network as a set of J directed links, individually identified as $j \in J$. Each link has capacity C_j. The term r is used to identify a specific path through the network. An individual flow is identified by the term s. The bandwidth assigned to a specific flow is identified by x_s, and the bandwidth from flow s assigned to path r is identified by y_r. We use the terminology $r \in s$ to denote the paths that are used by a specific flow and $r \in j$ to denote the paths that use link j. When we are referring to a specific path r, we use the expression $s(r)$ to denote the parent flow of the path.

The optimization program we use for a weighted α fair flow assignment is given by

$$\begin{array}{rl} \text{maximize} & \sum_{s\in S} w_s^\alpha \frac{x_s^{1-\alpha}}{1-\alpha} \\ \text{subject to} & \sum_{r\in s} y_r = x_s, \\ & \sum_{r\in j} y_r \le C_j \\ \text{over} & x, y > 0 \end{array}$$

The term w_s is a weight assigned to each flow, allowing the user to request that some flows be assigned proportionally more or less bandwidth than others.

Following [4], we develop the following set of update rules for solving this optimization problem using Lagrange multipliers μ_j for each capacity constraint:

$$y_r = \left(\left(\frac{w_{s(r)}}{x_{s(r)}} \right)^\alpha \cdot \frac{1}{\sum_{j\in r} \mu_j} \right)^{\frac{1}{1-q}} x_{s(r)} \tag{1}$$

$$\mu_j(t+1) = \mu_j(t) + \frac{1-q}{2} \mu_j(t) \left[\frac{\sum_{r\in j} y_r(t) - C_j}{C_j} \right] \tag{2}$$

$$x_s(t+1) = x_s(t) + \frac{1-q}{2(\alpha+q-1)} x_s(t) \cdot \left[\frac{\sum_{r\in s} y_r(t)^q - x_s(t)^q}{x_s(t)^q} \right] \tag{3}$$

The term q is a number close to but less than one used to ensure strict convexity in the problem.

Each of the update rules in equations (1), (2) and (3) can be implemented in parallel. In other words, all of the y_r values in (1) can be computed in parallel, then all of the μ_j values in (2) can be computed and so on. This prop-

HotSDN'14, August 22, 2014, Chicago, IL, USA.
ACM 978-1-4503-2989-7/14/08.
http://dx.doi.org/10.1145/2620728.2620762 .

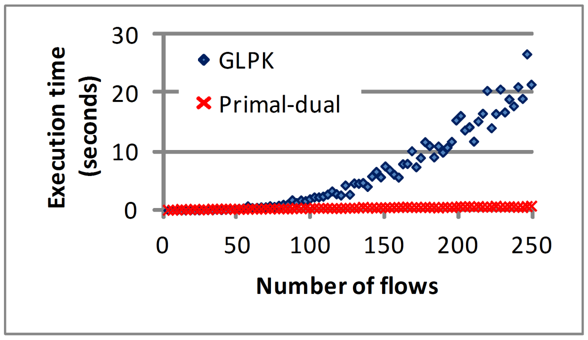

Figure 1: Max-min fair execution time

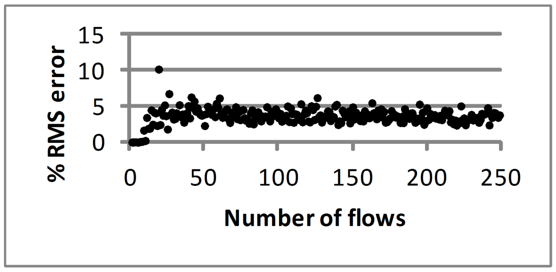

Figure 2: Max-min fair RMS error

erty makes it straightforward to implement the algorithm on massively parallel hardware.

3. PERFORMANCE RESULTS

We present performance results comparing the algorithm to a reference implementations for max-min fairness. The reference algorithm is Algorithm 8.3 from [3], implemented using the GNU linearing programming kit [1]. We have used BT's 21CN network topology comprising 106 nodes and 234 links within the United Kingdom as a reference for these simulations. Flows are generated using a pseudo-random number generator so that the end points for each flow are randomly selected. All flows are treated as elastic, so they will consume all network bandwidth available to them.

3.1 Workstation Results

Our workstation results were obtained on a 2.4GHz Xeon E5 processor. Our primal-dual algorithm is implemented in Java 7. We choose $q = 0.9$ and $\alpha = 4$ as an approximation for max-min fairness.

The workstation based results are shown in Figures 1 and 2. As expected, the execution time of the reference algorithm grows rapidly as larger problems require execution of a growing number of linear programs. Our algorithm shows a roughly linear increase in execution time with problem size. Choice of $q = 0.9$ and $\alpha = 4.0$ provides a good approximation of max-min fair, holding the root mean square error from the reference implementation at around 5%.

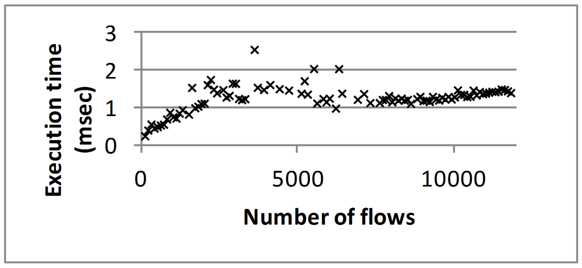

Figure 3: FPGA max-min fair execution time

3.2 FPGA Results

We have implemented our primal-dual algorithm in a Xilinx FPGA Virtex-7 XC7VX485T-3. The goal of the hardware implementation is to take advantage of the parallelism that can be found in the algorithm design to evaluate how we can improve the convergence time by executing concurrently as many operations as possible.

Our FPGA implementation consists of 32 parallel deep pipelines, each consisting of 180 stages which can produce one complete iteration of equations (1), (2) and (3) every clock cycle. With a core clock running at 200MHz, the FPGA produces 500G fixed point operations per second.

Performance results for our FPGA implementation are shown in Figure 3. This figure shows flow computation results for flow assignment with $q = 0.75$ and $\alpha = 4.0$ to provide an approximation of max-min fair flow assignment.

4. DISCUSSION

When we began work on this project, it was clear that solving flow assignment problems is a compute intensive task. Modern GPUs and FPGAs have lots of processing power, but it's only available in a highly parallel form. By exploiting the parallel structure of our algorithm, we have a solution that can run in millisecond times, making it fast enough to be part of the real time control loop in the network instead of an off-line tool.

The algorithm has proven simple to implement. Instead of the complex matrix factorization needed for interior point methods, we use straightforward vector operations for the update rules. The implementation team does not need to be versed in rules for matrix computations and can focus on identifying and eliminating performance bottlenecks.

5. REFERENCES

[1] Gnu linear programming kit. http://www.gnu.org/software/glpk. Accessed: 2014-03-06.

[2] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking*, 8(5):556–567, October 2000.

[3] M. Pioro and D. Medhi. *Routing, Flow, and Capacity Design in Communication and Computer Networks.* Morgan Kaufmann Publishers, 2004.

[4] T. Voice. Stability of multi-path dual congestion control algorithms. *IEEE/ACM Transactions on Networking*, 15(6):1231–1239, December 2007.