ParVarys: Parallelizing Coflow Scheduling in Haskell

Project Report - COMS 4995 Parallel Functional Programming

Etesam Ansari, Yunlan Li
Columbia University
{ea2905, yl4387}@columbia.edu

1 Introduction

We parallelized the Varys [4] coflow scheduling algorithm in Haskell. We utilized the data parallelism in Varys using Haskell's **`Eval`** monad and **`Strategy`**, and lazy data structures to parallelize Varys. Our parallel implementation achieved a max speed-up of 4x on a Macbook Pro M1 with 10 cores and 16GB memory.

In Section 2, we present some background on coflow scheduling and the Varys algorithm. In Section 3, we describe our sequential Haskell implementation of Varys, and then proceed to discuss how we parallelized it in Section 4, where we also present benchmark results that illustrate the effectiveness of our approaches and the pitfalls we encountered. Lastly, we summarize our takeaways from this project in Section 5.

2 Coflow Scheduling

Datacenter networks typically optimize for TCP *flow completion time*(FCT). However, nowadays, a datacenter application task could rarely be completed by sending a single request to another datacenter server. This has led to a mismatch between the network optimization objective(FCT) and application-level objective(Task Completion Time): minimizing FCT doesn't necessarily contribute to better task completion time. To address this challenge, a new abstraction called *coflow* is proposed: a collection of flows that share a common performance goal.

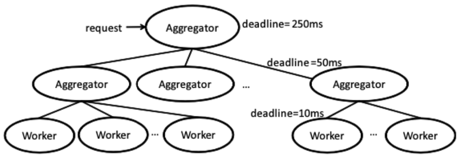


Figure 1: The partition/aggregate design pattern [3].

For example, web search workloads typically use a partition/aggregator model [3] where a user query will trigger multiple subtasks to search for results in each shard stored on different servers. The results from each shard are then aggregated to compute the final answer to be sent back to the user. In this example, all flows created for answering the user query would belong to the same coflow. Because latency observed by the end user is determined by the slowest of all flows of this coflow, it makes sense to optimize for this metric termed *coflow completion time*(CCT). By doing so, we could observe better application and end-user experience.

In the network community, there has been a large body of work to find a near-optimal coflow scheduling algorithm to minimize the average CCT. [1]

2.1 Offline Coflow Scheduling Problem (CSP)

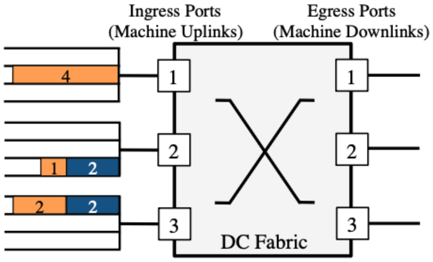


Figure 2: Coflow scheduling over a 3×3 datacenter fabric with three ingresss/egress ports. Flows in ingress ports are organized by destinations and color-coded by coflows - C_1 in orange/light and C_2 in blude/dark [4].

The offline coflow scheduling problem is defined as follows:

- the datacenter network fabric is abstracted into a single big switch consisting of m ingress ports (ToR switches)

---

[1] The average CCT of a collection of coflows.

and n egress ports (ToR switches)

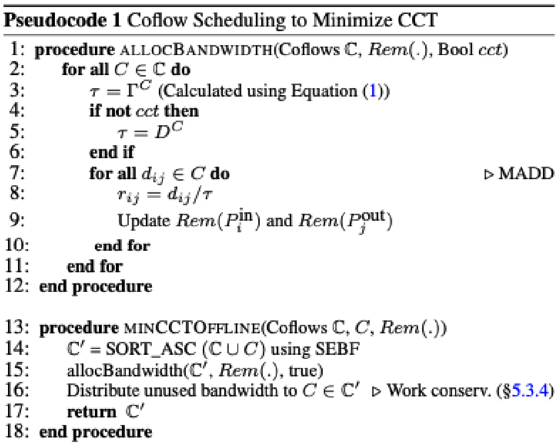
- assume all coflows arrive simultaneously at time $t = 0$, and the information about each coflow (number of flows, size, source and destination port of each flow) is all known [4].

The goal is to find a schedule for the coflows (order, rate) to minimize the average CCT. This problem is NP-hard (via reduction from concurrent open-shop scheduling problem).

2.2 Varys

Varys uses a Smallest-Effective-Bottleneck-First (SEBF) heuristic to produce a near-optimal ordering of coflows, and then perform rate allocations to optimize average CCT.

$$\Gamma^C = \max\left(\max_i \frac{\sum_i d_{ij}}{Rem\left(P_i^{in}\right)}, \max_j \frac{\sum_j d_{ij}}{Rem\left(P_j^{out}\right)}\right) \qquad (1)$$



```
Pseudocode 1 Coflow Scheduling to Minimize CCT
procedure allocBandwidth(Coflows $\mathbb{C}$, $Rem(.)$, Bool $cct$)
    for all $C \in \mathbb{C}$ do
        $\tau = \Gamma^C$ (Calculated using Equation (1))
        if not $cct$ then
            $\tau = D^C$
        end if
        for all $d_{ij} \in C$ do                                ▷ MADD
            $r_{ij} = d_{ij}/\tau$
            Update $Rem(P_i^{\text{in}})$ and $Rem(P_j^{\text{out}})$
        end for
    end for
end procedure

procedure minCCTOffline(Coflows $\mathbb{C}$, $C$, $Rem(.)$)
    $\mathbb{C}'$ = SORT_ASC ($\mathbb{C} \cup C$) using SEBF
    allocBandwidth($\mathbb{C}'$, $Rem(.)$, true)
    Distribute unused bandwidth to $C \in \mathbb{C}'$   ▷ Work conserv. (§5.3.4)
    return $\mathbb{C}'$
end procedure
```

Figure 3: Varys Offline Coflow Scheduling Algorithm [4].

In equation 1

- Γ^C represents the **shortest effective bottleneck** for coflow C
- d_{ij} represents the size of data that goes from ingress port i to egress port j
- $Rem(.)$ represents the remaining bandwidth of an ingress port or egress port
- P_i^{in} represents ingress port i, P_j^{out} represents egress port j

The Shortest job first (SJF) scheduling discipline is optimal for flow scheduling. SEBF can be viewed as an approximation for shortest job first in the context of coflow scheduling where a coflow consist of one or more flows instead of just one.

3 Sequential Haskell Implementation

3.1 CSP Representation

In datacenter networks, a centrallized controller running on comodity servers will be informed of the global view of network states, run Varys to perform coflow scheduling and finally disseminate the results to end switches. In an offline CSP problem, the network view consists of the state - switch ingress/egress bandwidth, and flow information - of all switches (see Figure 2).

In our implementation, a flow is abstracted into the `Flow` datatype, which holds the coflow id it belongs to, its flow size, and the egress port destination.

```
data Flow = Flow
  { coflowId      :: Int
  , size          :: Int
  , destinationId :: Int
  }
  deriving Show
```

We use `Switch` datatype to represent each ingress/egress port `iId :: Int`, which contains a number of `flows :: [Flow]` and has ingress/egress link rates specified by two `Int`.

```
data Switch = Switch
  { iId        :: Int
  , flows      :: [Flow]
  , iBandwidth :: Int
  , eBandwidth :: Int
  }
  deriving Show
```

With these, we define the input to Varys - i.e. a CSP problem - as the datatype `CSP` which is a datacenter-wide view of network state consisting of ingress and egress switch states.

```
data CSP = CSP
  { ingressSwitches :: [Switch]
  , egressSwitches  :: [Switch]
  }
  deriving Show
```

3.2 CSP Generator

We implement `generateProblem` to generate an offline CSP problem with `numIngress :: Int` ingress switches and `numEgress :: Int` egress switches. In addition, the function takes three specifications `RandomFlowSpec`, `RandomSwitchSpec`, and `RandomSwitchSpec` that defines the parameters for randomly generating the flows, ingress switches, and egress switches. In particular, for each ingress switch, we chose an `Int` in the range specified by `minFlows :: Int` and `maxFlows :: Int` of `ingressSwitchSpec :: RandomSwitchSpec` as the number of flows. Each flow has a size and coflow id randomly generated with min and max threshold specified by `RandomFlowSpec`.

```haskell
data RandomFlowSpec = RandomFlowSpec
  { minSwitchId :: Int
  , maxSwitchId :: Int
  , minCoflowId :: Int
  , maxCoflowId :: Int
  , minFlowSize :: Int
  , maxFlowSize :: Int
  }

data RandomSwitchSpec = RandomSwitchSpec
  { minFlows         :: Int
  , maxFlows         :: Int
  , ingressBandwidth :: Int
  , egressBandwidth  :: Int
  }

generateProblem
  :: RandomFlowSpec
  -> RandomSwitchSpec
  -> RandomSwitchSpec
  -> Int
  -> Int
  -> IO CSP
```

The flow and switch specification can be defined by setting relevant command line options:

```
% stack exec ParVarys-exe -- -h
ParVarys: Parallel Varys Coflow Scheduling Using SEBF

Usage: ParVarys-exe [-t|--type STRING] [-n|--coflows NUMBER]
                    [-i|--ingress NUMBER] [-e|--egress NUMBER]
                    [-s|--min-flow-size NUMBER] [-S|--max-flow-size NUMBER]
                    [-f|--min-switch-flows NUMBER]
                    [-F|--max-switch-flows NUMBER]
                    [-b|--ingress-bandwith NUMBER] [-B|--egress-bandwith NUMBER]
                    [--seed NUMBER]
  Generates an offline coflow scheduling problem, and uses the Varys Shortest
  Effective Bottlneck First heuristic to order the coflows.

Available options:
  -h,--help                Show this help text
  -t,--type STRING         Varys mode: seq, parMap, chunk (default: "parMap")
  -n,--coflows NUMBER      Number of coflows (default: 4000)
  -i,--ingress NUMBER      Number of ingress switches (default: 1000)
  -e,--egress NUMBER       Number of egress switches (default: 1000)
  -s,--min-flow-size NUMBER
                           Smallest flow size in bytes (default: 0)
  -S,--max-flow-size NUMBER
                           Largest flow size in bytes (default: 1000)
  -f,--min-switch-flows NUMBER
                           Minimum number of flows arriving at an ingress switch
                           (default: 0)
  -F,--max-switch-flows NUMBER
                           Maximum number of flows arriving at an ingress switch
                           (default: 5000)
  -b,--ingress-bandwith NUMBER
                           Ingress bandwidth (Gb/s) of a switch (default: 40)
  -B,--egress-bandwith NUMBER
                           Egress bandwidth (Gb/s) of a switch (default: 40)
  --seed NUMBER            Seed for global pseudo-random number generator
                           (default: 4995)
```

We use Haskell's standard pseudo-random number generator `StdGen` from the `System.Random` module. To reproduce a given CSP problem, one can specify an integer via the option `--seed` to seed the pseudo-random number generator.

3.3 Varys Controller

The Varys algorithm consists of two parts:

1. coflow ordering using the Shortest Effective Bottleneck First (SEBF) heuristic where the shortest effective bottleneck Γ of each coflow is computed with equation 1.
2. bandwidth allocation given the global coflow ordering in part 1.

For this project, we implemented part 1 and leave part 2 as future work.

The key insight is that the input to Varys `CSP` is a network-view centering around `Switch` entites, whereas Γ is a statistics derived from a coflow and switch bandwidth. This motivated us to first transform `CSP` into an intermediate representation that includes a coflow table `CoflowMap :: IntMap Coflow` and a switch bandwidth lookup table `BandwidthTable`.

```haskell
type Switch2Flow = IntMap.IntMap [Flow]
data FlowDirection = Ingress | Egress deriving (Eq, Show)
data Coflow =
  Coflow Int  -- coflow id
  [Flow]      -- all flows belonging to this coflow
  Switch2Flow -- flows grouped by ingress switch
  Switch2Flow -- flows grouped by egress switch
  deriving (Show)

type CoflowMap = IntMap.IntMap Coflow
type BandwidthTable = Map.Map (Int, FlowDirection) Int
```

This transformation is enabled by two functions that traverse the ingress and egress switches of `CSP` to incrementally build the desired `CoflowMap` and `BandwidthTable`.

```haskell
getSwitchBandwidth :: CSP -> BandwidthTable
toCoflows          :: CSP -> CoflowMap
```

With the abstraction of `Coflow`, computation of Γ becomes simple: the first fraction is a left-fold over the flows grouped by ingress switch with $Rem(.)$ looked up using the `BandwidthTable`, and the second fraction is a left fold over the flows grouped by egress switch. The computation of Γ is implemented by `getGamma` which takes in a `BandwidthTable`, `Coflow` and produces a `Rational` that represents the shortest effective bottleneck for the given coflow.

```haskell
getGamma :: BandwidthTable -> Coflow -> Rational
```

Lastly, we sort the coflows by their shortest effective bottlenck to produce a global coflow ordering schedule.

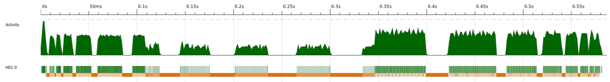
```haskell
-- Given a Coflow Scheduling Problem,
-- use the SEBF heuristic to order the Coflows.
--
-- Returns [(coflow id, shortest effective bottleneck)]
sebf :: CSP -> [(Int, Rational)]
sebf csp = Key.sort snd $ map f coflows
 where
  switchLinkRates = getSwitchBandwidth csp
  coflows         = IntMap.toList $ toCoflows csp
  f (cid, coflow) = (cid, getGamma switchLinkRates coflow)
```

4 Parallelizing Varys

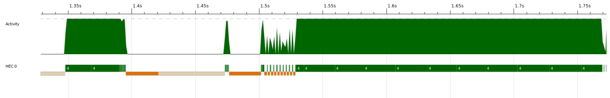
Unless otherwise specified, our experiments are run on an Apple Macbook Pro M1 with 10 cores and 16GB memory. We run both sequential and parallel version of our implementation of Varys on a CSP problem with 1000 ingress ports, 1000 egress ports, a maximum of 2000 coflows with a maximum flow size of 1000 bytes, and a maximum of 500 flows at each ingress port. In addition, both the ingress and egress bandwidth are set to 40Gbps. To ensure reproducibility of the generated CSP problem, we used the default seed of 4995.

We chose the parameters that generate the CSP problem so that it simulates a realistic workload in modern day datacenters where it's typical to have a fleet of thousands of servers, and thousands of coflows arriving per second.

4.1 Garbage Collector Optimization



(a) without garbage collector configuration



(b) with ghc options `-H8G -I10`

Figure 4: Threadscope visualizing event traces of sequential Varys running on a single HEC, with and without tuning GHC runtime options of garbage collector.

Figure 4(a) shows that when we run our sequential implementation of Varys with the default GHC settings, our application thread is frequently interleaved with the garbage collector thread (active throughout the duration of the program). This is explained by the scale of the problem we run our program against: thousands of switches and coflows whose data structures contains hundreds of flows, leaving a very large memory footprint.

Because of Amdahl's law, frequent interleaving garbage collection work would significantly limit the speedup we can achieve, we thus limit garbage collection by utilizing two GHC RTS options: `-H` for providing a suggested heap size for the garbage collector, and `-I` to increase the amount of time that must pass before an idle GC is performed [1]. We used 30s for the `-I` option, which is much larger than the time it takes for the program to run, and 8G (which is the maximum amount of memory typically available on the Macbook Pro M1) as the suggested heap size. Unless otherwise specified, these are the GHC RTS options we used for benchmarking.

This proved to be quite effective as can be seen from the comparison in Figure 4. Garbage collection disappears after 1.55s, yielding a 1.8x speedup (from 471.91ms to 265.69ms).

4.2 Data Parallelism

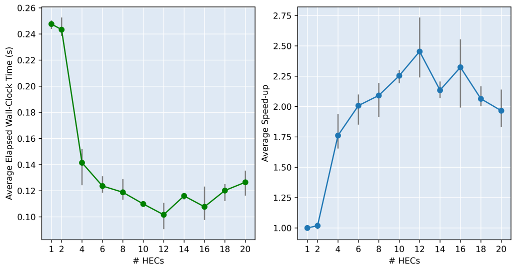


Figure 5: Average wall-clock time and average speed-up of Parvarys vs # HECs.

There are two main opportunities for data parallelism in our sequential implementation of Varys, which we exploit using Haskell's **`Eval`** Monad and **`Strategy`**. We manage to obtain a max speedup of 2.75x on 12 cores (see Figure 6). This is much lower than what we had expected and could partly be attributed to long pause of application threads due to garbage collection (see Figure 6).

Computation of Γ^C Recall equation 1, this represents the shortest effective bottleneck for a coflow C. Importantly, it is independent of $\Gamma^{C'}$ of other coflows, meaning that Γ of all coflows can be computed in parallel. We fully evaluate `f (cid, coflow) :: (Int, Rational)` to normal form using `rdeepseq`. To prevent spark overflow, we set a limit of the spark pool size with $maxParSparks = 4000$ using `parBuffer`.

```
parSebf :: CSP -> [(Int, Rational)]
parSebf csp = Key.sort
  snd
  (map f coflows `using` parBuffer maxParSparks rdeepseq)
 where
  switchLinkRates = parGetSwitchBandwidth csp
  coflows         = IntMap.toList $ parToCoflows csp
  f (cid, coflow) = (cid, getGamma switchLinkRates coflow)
```

Building CoflowMap from Ingress Switches This conversion is a left-fold over `ingressSwitches :: Switch` to produce a `coflowMap :: IntMap Coflow`. For each switch, the accumulator function iterates over each `flow :: Flow` of the switch, add updates the accumulated coflowMap by `cons` the flow to `flowsByIngress :: [Flow]` and `flowsByEgress :: [Flow]` of the coflow it belongs to. This accumulator function is associative, which allows us to transform it into a `map` and parallelize it. In particular, we

(1) split the ingress switches into chunks of 10, (2) build a partial coflowMap for each chunk, and (3) finally merge all the partial maps together using `unionsWith`. The construction of partial coflowMap for each chunk could then be parallelized using `parBuffer maxParSparks rdeepseq`.

```
data FlowDirection = Ingress | Egress deriving (Eq, Show)

instance NFData FlowDirection where
  rnf dir = dir `seq` ()

-- coflow: id  flows  flowsByIngress  flowsByEgress
data Coflow = Coflow Int [Flow] Switch2Flow Switch2Flow
  deriving Show

instance NFData Coflow where
  rnf (Coflow cid flows iFlows eFlows) =
    cid `seq` rnf flows `seq` rnf iFlows `seq` rnf eFlows

-- parallel version of toCoflows
parToCoflows :: CSP -> CoflowMap
parToCoflows csp = IntMap.unionsWith
  mergeCoflow
  (map f switchess `using` parBuffer maxParSparks rdeepseq)
 where
  f         = IntMap.unionsWith mergeCoflow . map (update
↪   IntMap.empty)
  switchess = chunksOf 10 $ ingressSwitches csp
```

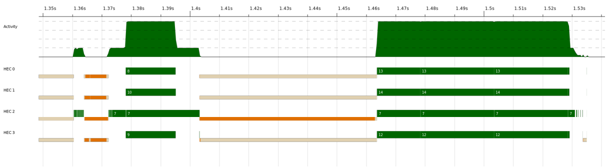


Figure 6: Threadscope visualizing event traces of parallel Varys running on 4 HECs, with data parallelism enabled.

The granularity of our data parallelism is fine enough to produce enough sparks to keep all HECs busy and is coarse enough to make the overhead of parallelization (spark creation, chunking, etc) worth it. For example, building a partial map from 10 ingress switches consist of a left-fold over thousands of flows. This is illustrated in Figure 6 where we can see work are spread evenly on all 4 HECs and constantly keeps the HECs busy. At time 1.39s to 1.41s, only HEC 2 busy, which is a result of the sequential `unionsWith` of partial coflowMaps in `parToCoflow :: CSP -> CoflowMap`. We mitigate this in Section 4.2 by using lazy maps. Lastly, near the end around 1.53s, again only HEC 3 is busy, this is due to sequentially sorting `(coflowId :: Int, gamma :: Rational)` pairs in `parSebf`. For thousands of coflows, this wasn't a bottleneck and thus we didn't attempt to parallelize it. However, when there are hundreds of thousands of coflows or more, this becomes the main performance bottleneck (see Figure 10 in Section 4.3).

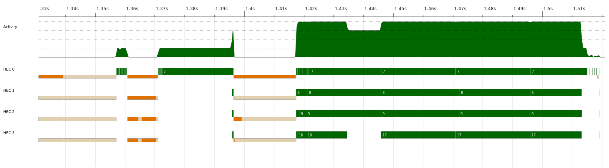


Figure 7: Threadscope visualizing event traces of parallel Varys running on 4 HECs, with data parallelism enabled and using lazy maps and chunking for building `BandwidthTable` and `CoflowMap`.

4.3 Lazy Coflow Transformation

To futher improve speed-up, we reduce the amount of sequential work performed in `parToCoflows :: CSP -> CoflowMap` by using lazy IntMap from `Data.List.IntMap.Lazy` module.

A lazy map is strict in its keys but lazy in its values [2]. This allows us to delay the computation of map values when they are needed, i.e. during the calculation of shortest-effective-bottleneck Γ^C for each coflow C. Since we parallelized the calculation of Γ^C, the evaluation of the map value of type $Coflow$ to normal form could thus be parallelized. With this change, we no longer spot the sequential merge of partial coflowMaps in threadscope (Figure 7), where only a single HEC is busy.

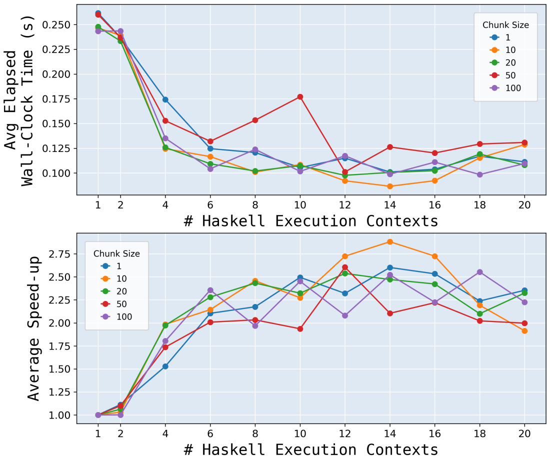


Figure 8: Comparison of average elapsed wall-clock time and speed-ups for building partial lazy `CoflowMap`s using chunks of N ingress switches and then perform a sequential union of these lazy IntMaps.

Initially, we thought chunk size we use to split ingress switches plays a big role in speedup, but experiment results say otherwise as Figure 8 shows. Initially, we thought that the amount of time the sequential **IntMap**`.unionsWith` is related to the number of maps it merges, so using a larger chunk size would take less time to complete. We later realized that this was incorrect since the time complexity of unions is determined by the number of keys, which would be the same regardless of the chunk size we choose. This explains the negligible difference in performance for different chunk sizes used.

4.4 Amdahl's Law Strikes

Figure 9: Threadscope visualizing event traces of parallel Varys running on 12 HECs, with all parallelization optimizations enabled.

Before parallelizing Varys, transforming the problem into coflowMap and calculating the shortest-effective-bottleneck for each coflow are the performance bottleneck. After parallelizing Varys, the more HECs we use, the less time it takes to perform the aforementioned calculations. What previously account for a small percentage of the total computation time now start to dominate and become the new bottleneck that prevent further speed-up (see Figure 9. This gives us an alternative view of Amdahl's Law, which is performance bottleneck shifts as we parallelize the algorithm, and eventually the sequential portion becomes the bottleneck that prevents further possibility for speed-up.

Initially, we thought Varys is straightforward to parallelize and easy to achieve linear speed-ups. However, as can be seen from our discussion so far, this is far from the case. Issues such as garbage collection due to the memory-intensive nature of the algorithm, map merging that's inherently sequential make Varys hard to parallelize. To make things worse, when there are hundreds of thousands coflows or more, sorting coflows by their shortest-effective-bottleneck now dominates the computation time as Figure 10 shows.

Figure 10: Threadscope visualizing event traces of parallel Varys running on 10 HECs solving a CSP problem of up to 500K coflows, 1K ingress ports and a maximum of 5000 flows per each ingress port.

5 Takeaways

Parallelizing Varys was harder than we expected and the amount of speed-ups achieved was also surprisingly far from our expectation going into the project. However, we found this project to be an interesting one and were able to take away something meaningful:

1. garbage collection can be really expensive for memory-intensive applications
2. alternative view of Amdahl's law is that performance bottleneck shifts
3. lazy data structures can be helpful for parallelization, and Haskell's paradigm of defining a structure for holding computation and a strategy to evaluate the computation is very elegant

References

[1] ghc runtime options. https://downloads.haskell.org/ghc/latest/docs/users_guide/runtime_control.html#rts-options-to-control-the-garbage-collector.

[2] Haskell lazy map. https://hackage.haskell.org/package/containers-0.6.6/docs/Data-Map-Lazy.html.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the*

2014 ACM Conference on SIGCOMM, SIGCOMM '14, page 443–454, New York, NY, USA, 2014. Association for Computing Machinery.

Appendix

src/Generator.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE NamedFieldPuns #-}

module Generator
  ( RandomFlowSpec(..)
  , RandomSwitchSpec(..)
  , Flow(..)
  , Switch(..)
  , CSP(..)
  , generateProblem
  ) where

import          Control.DeepSeq                ( NFData
                                              , rnf
                                              )
import          System.Random                  ( randomRIO )

data Flow = Flow
  { coflowId      :: Int
  , size          :: Int
  , destinationId :: Int
  }
  deriving Show

instance NFData Flow where
  rnf (Flow cid size dId) = cid `seq` size `seq` dId `seq` ()

data Switch = Switch
  { iId        :: Int
  , flows      :: [Flow]
  , iBandwidth :: Int
  , eBandwidth :: Int
  }
  deriving Show

-- Coflow Scheduling Problem
data CSP = CSP
  { ingressSwitches :: [Switch]
  , egressSwitches  :: [Switch]
  }
  deriving Show

data RandomFlowSpec = RandomFlowSpec
  { minSwitchId :: Int
  , maxSwitchId :: Int
  , minCoflowId :: Int
  , maxCoflowId :: Int
  , minFlowSize :: Int
  , maxFlowSize :: Int
```

```
  }

data RandomSwitchSpec = RandomSwitchSpec
  { minFlows         :: Int
  , maxFlows         :: Int
  , ingressBandwidth :: Int
  , egressBandwidth  :: Int
  }

-- FUNCTIONS:
-- Generates random Integer from lb to ub (inclusive? Yes)
generateRandomNum :: Int -> Int -> IO Int
generateRandomNum lb ub = do
  randomRIO (lb, ub)

generateFlows :: RandomFlowSpec -> Int -> IO [Flow]
generateFlows spec n = if n <= 0
  then do
    return []
  else do
    flows          <- generateFlows spec $ n - 1
    coflowId       <- generateRandomNum (minCoflowId spec) (maxCoflowId spec)
    egressSwitchId <- generateRandomNum (minSwitchId spec) (maxSwitchId spec)
    flowSize       <- generateRandomNum (minFlowSize spec) (maxFlowSize spec)

    return $ Flow coflowId flowSize egressSwitchId : flows

generateSwitches
  :: RandomFlowSpec -> RandomSwitchSpec -> Int -> Int -> IO [Switch]
generateSwitches flowSpec switchSpec minId maxId = if maxId - minId < 0
  then do
    return []
  else do
    numOfFlows <- generateRandomNum (minFlows switchSpec) (maxFlows switchSpec)
    flows      <- generateFlows flowSpec numOfFlows
    let switch = Switch minId
                        flows
                        (ingressBandwidth switchSpec)
                        (egressBandwidth switchSpec)

    switches <- generateSwitches flowSpec switchSpec (minId + 1) maxId
    return $ switch : switches

generateProblem
  :: RandomFlowSpec
  -> RandomSwitchSpec
  -> RandomSwitchSpec
  -> Int
  -> Int
  -> IO CSP
generateProblem flowSpec ingressSwitchSpec egressSwitchSpec numIngress numEgress
  = do
    let (minIngressId, maxIngressId) = (1, numIngress)
        (minEgressId , maxEgressId ) = (numIngress + 1, numIngress + numEgress)
```

```

    iSwitches <- generateSwitches flowSpec
                                  ingressSwitchSpec
                                  minIngressId
                                  maxIngressId
    eSwitches <- generateSwitches flowSpec
                                  egressSwitchSpec
                                  minEgressId
                                  maxEgressId

    return $ CSP iSwitches eSwitches
```

src/Controller.hs

```haskell
{-# LANGUAGE NamedFieldPuns #-}

module Controller
  ( Coflow(..)
  , toCoflows
  , parToCoflows
  , getSwitchBandwidth
  , getGamma
  , sebf
  , parSebf
  ) where

import           Control.DeepSeq                ( NFData
                                                , rnf
                                                )
import           Control.Parallel.Strategies    ( parBuffer
                                                , rdeepseq
                                                , using
                                                )
import qualified Data.IntMap.Lazy              as IntMap
import qualified Data.List.Key                 as Key
import           Data.List.Split                ( chunksOf )
import qualified Data.Map.Lazy                 as Map
import           Data.Maybe                     ( fromMaybe )
import           Data.Ratio                     ( (%) )

import           Generator                      ( CSP(..)
                                                , Flow(..)
                                                , Switch(..)
                                                )


data FlowDirection = Ingress | Egress deriving (Eq, Show)

instance NFData FlowDirection where
  rnf dir = dir `seq` ()

instance Ord FlowDirection where
  a <= b = case (a, b) of
    (Egress, Ingress) -> False
    _                 -> True

-- coflow: id  flows  flowsByIngress  flowsByEgress
data Coflow = Coflow Int [Flow] Switch2Flow Switch2Flow
  deriving Show

instance NFData Coflow where
  rnf (Coflow cid flows iFlows eFlows) =
    cid `seq` rnf flows `seq` rnf iFlows `seq` rnf eFlows


type Switch2Flow = IntMap.IntMap [Flow]
```

```haskell
type CoflowMap = IntMap.IntMap Coflow
type BandwidthTable = Map.Map (Int, FlowDirection) Int


maxParSparks :: Int
maxParSparks = 4000

updateMap :: Int -> Flow -> Switch2Flow -> Switch2Flow
updateMap k v = IntMap.alter f k
 where
  f pv = case pv of
    Nothing -> Just [v]
    Just vs -> Just $ v : vs

addFlow :: CoflowMap -> (Int, Flow) -> CoflowMap
addFlow currMap (ingressPort, flow) = IntMap.alter f (coflowId flow) currMap
 where
  egressPort = destinationId flow
  f val = case val of
    Nothing -> Just $ Coflow (coflowId flow)
                             [flow]
                             (IntMap.singleton ingressPort [flow])
                             (IntMap.singleton egressPort [flow])
    Just (Coflow cid coflow flowsByISwitch flowsByESwitch) -> Just $ Coflow
      cid
      (flow : coflow)
      (updateMap ingressPort flow flowsByISwitch)
      (updateMap egressPort flow flowsByESwitch)

update :: CoflowMap -> Switch -> CoflowMap
update currMap switch =
  foldl addFlow currMap $ zip (repeat $ iId switch) (flows switch)

-- Assumes that the coflowId of the two coflows passed in are the same
mergeCoflow :: Coflow -> Coflow -> Coflow
mergeCoflow (Coflow cid flows ingress egress) (Coflow _ flows' ingress' egress')
  = Coflow cid
           (flows ++ flows')
           (IntMap.unionWith (++) ingress ingress')
           (IntMap.unionWith (++) egress egress')

toCoflows :: CSP -> CoflowMap
toCoflows csp = foldl update IntMap.empty $ ingressSwitches csp

parToCoflows :: CSP -> CoflowMap
parToCoflows csp = IntMap.unionsWith
  mergeCoflow
  (map f switchess `using` parBuffer maxParSparks rdeepseq)
 where
  f          = IntMap.unionsWith mergeCoflow . map (update IntMap.empty)
  switchess = chunksOf 10 $ ingressSwitches csp

getSwitchBandwidth :: CSP -> BandwidthTable
getSwitchBandwidth csp = Map.fromList $ concatMap f switches where
```

```haskell
  f (Switch sid _ iBw eBw) = [((sid, Ingress), iBw), ((sid, Egress), eBw)]
  switches = ingressSwitches csp ++ egressSwitches csp

parGetSwitchBandwidth :: CSP -> BandwidthTable
parGetSwitchBandwidth csp = Map.fromList $ concat
  (map (concatMap f) switchess `using` parBuffer maxParSparks rdeepseq)
 where
  f (Switch sid _ iBw eBw) = [((sid, Ingress), iBw), ((sid, Egress), eBw)]
  switchess = chunksOf 20 $ ingressSwitches csp ++ egressSwitches csp

getGamma :: BandwidthTable -> Coflow -> Rational
getGamma bwTbl (Coflow _ _ ingressFlows egressFlows) = max
  (maximum ingressTimes)
  (maximum egressTimes)
 where
  sumFlows :: (Int, [Flow]) -> (Int, Int)
  sumFlows (switchId, flows) = (switchId, foldl (\a el -> a + size el) 0 flows)

  calcTime :: FlowDirection -> (Int, Int) -> Rational
  calcTime flowDir (switchId, flowSize) = fromIntegral flowSize
    % fromIntegral bandwidth
    where bandwidth = fromMaybe 0 $ Map.lookup (switchId, flowDir) bwTbl

  ingressTimes = map (calcTime Ingress . sumFlows) $ IntMap.toList ingressFlows
  egressTimes  = map (calcTime Egress . sumFlows) $ IntMap.toList egressFlows


-- Given a Coflow Scheduling Problem, use Shortest Effective Bottleneck First
-- heuristic to order the Coflows.
--
-- Returns [(coflow id, effective bottleneck)]
sebf :: CSP -> [(Int, Rational)]
sebf csp = Key.sort snd $ map f coflows
 where
  switchLinkRates = getSwitchBandwidth csp
  coflows         = IntMap.toList $ toCoflows csp
  f (cid, coflow) = (cid, getGamma switchLinkRates coflow)

-- Parallel version of sebf
parSebf :: CSP -> [(Int, Rational)]
parSebf csp = Key.sort
  snd
  (map f coflows `using` parBuffer maxParSparks rdeepseq)
 where
  switchLinkRates = parGetSwitchBandwidth csp
  coflows         = IntMap.toList $ parToCoflows csp
  f (cid, coflow) = (cid, getGamma switchLinkRates coflow)
```

app/Main.hs

```haskell
{-# LANGUAGE NamedFieldPuns #-}

import           Control.DeepSeq                 ( force )
import           Control.Exception               ( evaluate )
import           Control.Monad                   ( join )
import           Formatting                      ( fprintLn )
import           Formatting.Clock                ( timeSpecs )
import           Generator                       ( RandomFlowSpec(..)
                                                 , RandomSwitchSpec(..)
                                                 , generateProblem
                                                 )
import           Options.Applicative
import           System.Clock
import           System.Exit                     ( die )
import           System.Random                   ( mkStdGen
                                                 , setStdGen
                                                 )

import           Controller                      ( parSebf
                                                 , sebf
                                                 )

-- Arg Parser template adapted from:
--  https://ro-che.info/articles/2016-12-30-optparse-applicative-quick-start
main :: IO ()
main = join . customExecParser (prefs showHelpOnError) $ info
  (helper <*> parser)
  (  fullDesc
  <> header "ParVarys: Parallel Varys Coflow Scheduling Using SEBF "
  <> progDesc
       (  "Generates an offline coflow scheduling problem, and uses the "
       ++ "Varys Shortest Effective Bottlneck First heuristic to order "
       ++ "the coflows."
       )
  )
 where
  parser :: Parser (IO ())
  parser =
    work
      <$> strOption
            (  long "type"
            <> short 't'
            <> metavar "STRING"
            <> help "Varys mode: seq, parMap"
            <> value "parMap"
            <> showDefault
            )
      <*> option
            auto
            (  long "coflows"
            <> short 'n'
            <> metavar "NUMBER"
```

```
          <> help "Number of coflows"
          <> value 4000
          <> showDefault
          )
    <*> option
          auto
          (  long "ingress"
          <> short 'i'
          <> metavar "NUMBER"
          <> help "Number of ingress switches"
          <> value 1000
          <> showDefault
          )
    <*> option
          auto
          (  long "egress"
          <> short 'e'
          <> metavar "NUMBER"
          <> help "Number of egress switches"
          <> value 1000
          <> showDefault
          )
    <*> option
          auto
          (  long "min-flow-size"
          <> short 's'
          <> metavar "NUMBER"
          <> help "Smallest flow size in bytes"
          <> value 0
          <> showDefault
          )
    <*> option
          auto
          (  long "max-flow-size"
          <> short 'S'
          <> metavar "NUMBER"
          <> help "Largest flow size in bytes"
          <> value 1000
          <> showDefault
          )
    <*> option
          auto
          (  long "min-switch-flows"
          <> short 'f'
          <> metavar "NUMBER"
          <> help "Minimum number of flows arriving at an ingress switch"
          <> value 0
          <> showDefault
          )
    <*> option
          auto
          (  long "max-switch-flows"
          <> short 'F'
          <> metavar "NUMBER"
```

```
            <> help "Maximum number of flows arriving at an ingress switch"
            <> value 5000
            <> showDefault
            )
      <*> option
            auto
            (  long "ingress-bandwith"
            <> short 'b'
            <> metavar "NUMBER"
            <> help "Ingress bandwidth (Gb/s) of a switch"
            <> value 40
            <> showDefault
            )
      <*> option
            auto
            (  long "egress-bandwith"
            <> short 'B'
            <> metavar "NUMBER"
            <> help "Egress bandwidth (Gb/s) of a switch"
            <> value 40
            <> showDefault
            )
      <*> option
            auto
            (  long "seed"
            <> metavar "NUMBER"
            <> help "Seed for global pseudo-random number generator"
            <> value 4995
            <> showDefault
            )


work
  :: String
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> Int
  -> IO ()
work mode numCoflows numIngress numEgress minFlowSize maxFlowSize minFlows maxFlows ingressBandwidth egressBandwidth seed
  = do
    sebfImpl <- case mode of
      "seq"    -> return sebf
      "parMap" -> return parSebf
      _ ->
        die
          $  "Unrecognized varys mode: "
```

```haskell
          ++ "expect one of seq, parMap, got "
          ++ show mode

    let flowSpec = RandomFlowSpec { minSwitchId = numIngress + 1
                                  , maxSwitchId = numIngress + numEgress
                                  , minCoflowId = 1
                                  , maxCoflowId = numCoflows
                                  , minFlowSize
                                  , maxFlowSize
                                  }

        ingressSwitchSpec = RandomSwitchSpec { minFlows
                                             , maxFlows
                                             , ingressBandwidth
                                             , egressBandwidth
                                             }

        egressSwitchSpec = RandomSwitchSpec { minFlows         = 0
                                            , maxFlows         = 0
                                            , ingressBandwidth
                                            , egressBandwidth
                                            }

    -- seed the global pseudo-random number generator
    -- for reproducibility of CSP problems
    setStdGen $ mkStdGen seed
    problem <- generateProblem flowSpec
                               ingressSwitchSpec
                               egressSwitchSpec
                               numIngress
                               numEgress

    start       <- getTime Monotonic
    coflowOrder <- evaluate $ force $ sebfImpl problem
    end         <- getTime Monotonic

    print coflowOrder
    putStr "Calculation Time: "
    fprintLn timeSpecs start end
```

test/Spec.hs

```haskell
import           Data.Ratio                     ( (%) )
import           System.Random                  ( mkStdGen
                                                , setStdGen
                                                )
import           Test.HUnit

import           Controller                     ( parSebf
                                                , sebf
                                                )
import           Generator                      ( CSP(..)
                                                , Flow(..)
                                                , RandomFlowSpec(..)
                                                , RandomSwitchSpec(..)
                                                , Switch(..)
                                                , generateProblem
                                                )

testSebf1 :: Test
testSebf1 = TestCase
  (do
    let csp = CSP
          { ingressSwitches =
            [ Switch
              { iId        = 1
              , iBandwidth = 1
              , eBandwidth = 1
              , flows = [Flow { coflowId = 1, size = 4, destinationId = 5 }]
              }
            , Switch
              { iId        = 2
              , iBandwidth = 1
              , eBandwidth = 1
              , flows = [ Flow { coflowId = 1, size = 1, destinationId = 6 }
                        , Flow { coflowId = 2, size = 2, destinationId = 6 }
                        ]
              }
            , Switch
              { iId        = 3
              , iBandwidth = 1
              , eBandwidth = 1
              , flows = [ Flow { coflowId = 1, size = 2, destinationId = 4 }
                        , Flow { coflowId = 2, size = 2, destinationId = 4 }
                        ]
              }
            ]

          , egressSwitches  = [ Switch { iId        = n
                                       , iBandwidth = 1
                                       , eBandwidth = 1
                                       , flows      = []
                                       }
                              | n <- [4 .. 6]
```

```
                              ]
          }

    assertEqual "" [(2, 2 % 1), (1, 4 % 1)] $ sebf csp
  )

testSebf2 :: Test
testSebf2 = TestCase
  (do
    let csp = CSP
          { ingressSwitches =
            [ Switch
              { iId        = 1
              , iBandwidth = 2
              , eBandwidth = 1
              , flows = [Flow { coflowId = 1, size = 4, destinationId = 5 }]
              }
            , Switch
              { iId        = 2
              , iBandwidth = 1
              , eBandwidth = 1
              , flows = [ Flow { coflowId = 1, size = 1, destinationId = 6 }
                        , Flow { coflowId = 2, size = 2, destinationId = 4 }
                        ]
              }
            , Switch
              { iId        = 3
              , iBandwidth = 1
              , eBandwidth = 1
              , flows = [ Flow { coflowId = 1, size = 2, destinationId = 4 }
                        , Flow { coflowId = 2, size = 2, destinationId = 4 }
                        ]
              }
            ]

          , egressSwitches  =
            [ Switch { iId = 4, iBandwidth = 1, eBandwidth = 1, flows = [] }
            , Switch { iId = 5, iBandwidth = 1, eBandwidth = 4, flows = [] }
            , Switch { iId = 6, iBandwidth = 1, eBandwidth = 1, flows = [] }
            ]
          }

    assertEqual "" [(1, 2 % 1), (2, 4 % 1)] $ sebf csp
  )

-- validate the correctness of parallel implementation using
-- the sequential implementation of SEBF
testParSebf1 :: Test
testParSebf1 = TestCase
  (do
    let seed     = 91845734
        flowSpec = RandomFlowSpec { minSwitchId = 201
                                  , maxSwitchId = 400
                                  , minCoflowId = 1
```

```
                                  , maxCoflowId = 1000
                                  , minFlowSize = 0
                                  , maxFlowSize = 100
                                  }
        ingressSpec = RandomSwitchSpec 0 100 40 40
        egressSpec  = RandomSwitchSpec 0 0 40 40

    setStdGen $ mkStdGen seed
    problem <- generateProblem flowSpec ingressSpec egressSpec 200 200
    assertEqual "" (sebf problem) (parSebf problem)
  )


main :: IO Counts
main = runTestTT $ TestList
  [ "SEBF Sequential (varys_paper_fig1)" ~: testSebf1
  , "SEBF Sequential (variable_link_rates)" ~: testSebf2
  , "SEBF Parallel   (parMap)" ~: testParSebf1
  ]
```