Minimizing Coflow Completion Time
Final Report

Christopher Zhu and Thomas Dedinsky

2020-08-12

Abstract

This paper details the implementation and execution of an approximation algorithm designed to minimize coflow completion time. The paper 'Sincronia: Near-Optimal Network Design for Coflows' (2018) by Saksham Agarwal et al is the basis for all of the subject matter in this report. The offline version of this algorithm, where all jobs are known at the start, is tested first with a toy example followed by generated testbed using a workload generator created for Sincronia. The results from Sincronia are then compared with an established algorithm.

1. Problem Explanation

The algorithm presented in this paper, 'Sincronia'[1], uses the following model:

i. The datacenter fabric is abstracted to a singular switch

ii. The datacenter fabric can sustain 100% throughput

iii. The only areas of congestion are the ingress and egress queues of the datacenter fabric 'switch'

The coflow problem Sincronia is trying to solve involves multiple coflows c and their completion time. A coflow is a combination of multiple flows split across multiple ingress ports i which each want to go to a specified egress port j. Each coflow flow c_k received by ingress i, which will be referred to as a job, has a specific id k, a set duration t, as well as a specific egress j it needs to be processed by. All ingresses and egresses are connected to a master switch, which will handle scheduling and can connect an ingress to an

appropriate egress to process a specified job. The number of ingress ports is equal to the number of egress ports, which means that, at any given instance, all ports can be active and transferring flow. In the offline version, all coflows arrive at time $t = 0$, so the coflow completion time (CCT_c) is defined as the completion time of its last job c_k. The objective is to minimize the weighted average coflow completion time, $(\sum_c w_c \times \mathrm{CCT}_c)/n$.

2. Implementation

2.1. Multi-server Communication

The main code works using three different types of processes: clients, frontend (FE) node, and backend (BE) nodes. Each of these were implemented in Java 1.11, but since they communicate using Apache Thrift 0.13.0, they could all be in a different language and still function fine. Apache Thrift is used in large infrastructures with multi-server communication and thus is a good fit for this type of datacenter fabric switch implementation. A convenient build script was made which also gives sample commands to run this implementation.

For the client processes, the code initially starts by reading out a schedule text file. The format used by this file is used by the coflow generator provided by the same team which made the paper [2]. This dictates, for each coflow, how many different flows it contains, its arrival time, and for each flow, its ingress and egress ports and size. It builds up a list of jobs for each ingress port, and then assigns each of these lists to a client thread. It runs all of these threads, which simply connect to the FE node and send it the schedule for that ingress port. The client will let the FE node know ahead of time how many ingresses (and thus egresses) it should expect.

For the BE node, upon boot it tries to connect to the FE node. If the node isn't available, it will continue polling until it can connect. Once it does, it connects and waits on standby for any available jobs in its service handler. Since this is just a test, when it receives a list of jobs, it doesn't do any additional processing besides printing out diagnostic information about the job and its execution. Also, multiple BE nodes from multiple computers can be connected at once to handle a large workload and is supported in the code, but for the purposes of this project, only one multi-threaded node was

used.

The FE node is where most of the work happens. For managing connections, it takes care of BE node management. For the purposes of this experiment, it only does limited error checking to see if a BE node has gone down, and handles reconnects and concurrency management. It makes no attempt to reissue a schedule to an egress if a node crashes during processing (which isn't expected due to it just being print statements), and if not enough egresses are available it will just give each egress additional schedules.

However, the actual Sincronia implementation is the most vital part of this project. Once it receives a list of jobs from one of the client "ingresses", it tabulates the coflow ids and the egresses, as well as adding the list of jobs to the list of ingress schedules received. Once it has received all of the schedules, it requests an appropriate amount of BE node threads to use and then starts computing the schedules for each of these threads. It will send them off to each of the threads once computation is complete.

There are two main parts to the implementation. The first part is calculating the ordering of each coflow. Having the list of all of the ingresses, egresses, and ids, I make arrays for all three of them for easier indexing and dictionaries for the latter two so I know what index an id since the job object only has the id. It then runs the Bottleneck-Select-Scale-Iterate (BSSI) Algorithm. This will iterate through each ingress, see which one has the remaining processing time out of unassigned jobs, and investigate that ingress. From there, it will pick which flow set, jobs at that port from the same coflow, has the maximum weighted duration in that ingress and assign that flow set's coflow to the last unassigned coflow in the ordering (it assigns from last to first). It then update the weights of each remaining coflow, subtracting the weight of the chosen coflow multiplied by the ratio of the remaining coflow's job's duration (if one exists at that ingress) and the selected job's duration. It will iterate through this until all coflows are assigned an ordering. All ties are broken up by largest coflow id.

The second part is actually scheduling the jobs. The FE node works like a switch, as in only one ingress port can be connected to one egress port at any given time. So, each ingress schedule is sorted by the previously determined job ordering, but each ingress is sorted by the next available time as

a primary metric first, and then ties are broken up by whatever the ordering of the job which is next in each of the ingress' job lists. Until all ingress job lists are completely scheduled, a loop occurs where the next available ingress' (from the queue) schedule is iterated until it can find a job with a free egress (both ingress and egress availability times are being kept track of). Once it does, it will schedule that job for the egress and update both the ingress and egress availability time. If it doesn't, it'll update the ingress' availability time to the time of the next available egress it can schedule a job to. Either way, the ingress is put back into the ingress queue based on the two previously mentioned metrics. Once schedules are completed, the average weighted coflow completion time is computed.

A lot of implementation details were skimmed over. First of all, the time duration of a job is actual both an integer value as well an epsilon. This is modelled after the original schedule used in the paper, which was also used for toy testing. The other examples do not use this feature. So, while the Job class used had two separate integer values for the integer and epsilon, as well as the egress and coflow id, the weighting algorithm had to deal with division and fractions. So, an EpsilonFraction class was created for handling this, with methods for creation, addition, multiplication, division, and comparison. Additionally, the job and ingress sorts done at the start of the job scheduling algorithms were done with Java comparators as opposed to manual sorting algorithms like merge sort. The ones done after each iteration were done using one iteration of manual linear sort, as only one element changed between iterations.

The only minor sub-optimality with this implementation of offline Sincronia is the case where two ingresses have the same availability time, then ingress A has a first job that has priority ordering but its second job should be ordered after ingress B's first job, then the first job of ingress A cannot be executed because the requested egress is currently busy. In an ideal case, ingress B's first job should then be able to be scheduled. However, this would require an additional complexity penalty at each iteration, because in the worst case scenario every single ingress would be tied and thus each of their schedules would have to be iterated through until their first available job (potentially no job!) is available, and then comparing all of these available jobs to see which is the first job that should be scheduled. And then this comparison happens m more times for m remaining ingresses for that time

slot. For a given job scheduling, this would bump up the complexity from O(q), q being the max length of an ingress' schedule, to O(max($mlog(m)$, mq)). The paper does not recommend a tiebreaker method like with the ordering calculation, so it is unclear which implementation was intended.

In terms of runtime analysis, the weight calculation is O($max(np, nm)$, where n is the number of coflows and p is the number of jobs. This is because it iterates through each coflow to determine its weight and needs to check each job and each ingress to determine which coflow is next last. The schedule assignment is O($max(mqlog(q), mlog(m), pmq, pm^2)$). First it sorted all the jobs for each ingress schedule, then it sorts the ingress schedules, and this is responsible for the first two statements. Then it will loop until no jobs remain and check if the next available ingress can send a job in decreasing priority to an egress, will schedule the job if it can or update its available time if it can't, and place itself back in the list of ingresses. The absolute worst case scenario that, for every single job assignment, we need to go through each ingress schedule who want the same egress, iterate to the end to find out all of the egresses are busy, put it back in line, and go to the next ingress. Once all of the ingresses have had their availability time updated, one ingress takes the egress, and the rest have to update their availability time again. That's pm iterations with each iteration checking an ingress' schedule and a list of ingresses.

2.2. Single Process Mode

For ease of use, a program which uses all of the helper functions for the processes described above was created. It reads the input file, calculates the schedule, and prints the job ordering, average weighted coflow completion time, and each schedule. It does not require Apache Thrift to use.

2.3. Brute Force Solver

To compare the approximate schedule with the actual optimal schedule and the difference in optimality, a brute force solver was made to compute every single possible schedule to find the optimal schedule. This was a complex enough procedure to implement and optimize that a multi-paragraph discussion of its implementation will be detailed below.

First of all, all of the schedules and logic are sorted from an egress-sorted schedule. This is different than the ingress-sorted schedule that the approximation algorithm uses, but makes no immediate difference on runtime length.

However, a difference from the approximation algorithm is that the schedule is directly modified to include the array index of the ingress and coflow id to allow for faster execution of job operations.

Next, we recursively generate each possible schedule ordering. This starts by finding every single job permutation for egress 1, and then for each permutation, finding every single job permutation for egress 2, all the way until the final egress. Then, with a unique job permutation assembled, it will start analyzing that particular schedule ordering.

For the purposes of this discussion, an action in the execution of a schedule is assigning the next flow to go from a particular ingress to a particular egress. This flow has priority over any future flows assigned from an action. This comes up in particular if two egresses are looking to take from the same ingress for their next action. Whichever one get priority will not have to wait for the other egress.

The time efficiency problem with taking an action is that different orders of actions can result in different schedules despite the same job ordering. This is evident with the above example. After each action, the solver would then have to analyze the alternate schedules created by invoking an action for each one of the remaining egresses. This would be very inefficient.

Luckily, there are two optimizations that we can make which involve taking only one action, thereby not increasing the number of possibilities. The first is if an egress is requesting an ingress which will not be used by any other egress going forward. The second is if an egress is requesting an ingress which cannot be requested by any egress before the job caused by this action would be completed. Both of these action types are free to occur whenever, and the solver will only diverge in paths once a full iteration of egress checks occurs where neither of these two optimizations are present.

The worst-case runtime of this solver is staggeringly large. Let us use m ingresses/egress, and p jobs, where q is the most number of jobs at an ingress/egress. This means we have to go through q permutations within q permutations recursively for each egress, and for each unique combination we may diverge m ways for all p jobs. This means the solver is in magnitude of $\mathrm{O}((q!)^m * m^p)$.

There are two versions of this brute force solver, one is iterative and one is recursive. However, both versions of this encounter memory issues. The iterative version encounters heap overflow for large schedules, whereas the recursive version can encounter heap or stack overflow for large schedules. Unfortunately, this means that it cannot be used practically for the big schedules desired to be tested.

3. Computational Results

3.1. Toy Example

To test the accuracy of the Java implementation, the scenario from Figure 1 in Agarwal's paper [1] was recreated and executed. The following two figures are the verbose output from the FE node and BE node which perfectly mirror the schedule created by Sincronia in Figure 2b of Agarwal's paper [1], obtaining an average wCCT of 2.6. The optimal value, as shown in the paper if job 1 obtains last priority, is 2.4. Note that the brute force solver also obtained this optimal average wCCT.

Figure 1: Console output from the FE node.

```
0 [main] INFO FENode  - Launching FE node on port 10123
2817 [pool-1-thread-1] INFO FENodeServiceHandler  - Added client tdedinsk:10124 with 4 thread(s).
5524 [pool-1-thread-5] INFO CalculateSchedules  - Order of jobs: 2; 3; 4; 1; 5;
wCCT = 13+5e
coflows = 5
average wCCT = 2.6
5567 [pool-1-thread-5] INFO FENodeServiceHandler  - Calling 4 backend node thread(s)
5568 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Locked client 0 for processing
5570 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Locked client 1 for processing
5570 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Locked client 2 for processing
5570 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Locked client 3 for processing
5635 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Unlocked client 0 from processing
5636 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Unlocked client 1 from processing
5636 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Unlocked client 2 from processing
5636 [pool-1-thread-5] DEBUG FENodeServiceHandler  - Unlocked client 3 from processing
```

Figure 2: Console output from the BE node.

```
0 [main] INFO BENode  - Launching BE node on port 10124 at host tdedinsk
Initialized client
1051 [main] INFO BENode  - tdedinsk
1071 [main] INFO BENode  - Connected to Frontend
3990 [pool-1-thread-5] INFO CalculateSchedules  - BENode b (t = 0 e0): Job 3 processed in 2 e1
3993 [pool-1-thread-5] INFO CalculateSchedules  - BENode b (t = 2 e1): Job 1 processed in 1 e0
3993 [pool-1-thread-2] INFO CalculateSchedules  - BENode c (t = 0 e0): Job 4 processed in 2 e1
3993 [pool-1-thread-2] INFO CalculateSchedules  - BENode c (t = 2 e1): Job 1 processed in 1 e0
3993 [pool-1-thread-2] INFO CalculateSchedules  - BENode c (t = 3 e1): Job 1 processed in 1 e0
3989 [pool-1-thread-3] INFO CalculateSchedules  - BENode a (t = 0 e0): Job 2 processed in 2 e1
3993 [pool-1-thread-3] INFO CalculateSchedules  - BENode a (t = 2 e1): Job 1 processed in 1 e0
3994 [pool-1-thread-3] INFO CalculateSchedules  - BENode a (t = 3 e1): Job 1 processed in 1 e0
3989 [pool-1-thread-4] INFO CalculateSchedules  - BENode d (t = 0 e0): Job 1 processed in 1 e0
3993 [pool-1-thread-5] INFO CalculateSchedules  - BENode b (t = 3 e1): Job 1 processed in 1 e0
3994 [pool-1-thread-4] INFO CalculateSchedules  - BENode d (t = 1 e0): Job 5 processed in 2 e1
3994 [pool-1-thread-4] INFO CalculateSchedules  - BENode d (t = 3 e1): Job 1 processed in 1 e0
```

3.2. Experimental Setup

Four workload testbeds were generated using the Sincronia project's 'Workload Generator' project [2]. The Sincronia paper mentions in Section 5.1 that the baseline configuration for the generated workloads was 2000 coflows, 0.9 network load, and a time horizon reset after 8 epochs. The time horizon reset is not used in our implementation of Sincronia. The workload testbeds were generated with the following values for non-variable parameters: 500 coflows, 20 maximum coflow width, 0.9 network load, 0.5 port contention, uniform source frequency and destination data distributions, and 150 ports. The reason for 150 ports is to mimic the Facebook production log trace, 'Coflow Benchmark' [3], which the workload generator is based on. The reasons for 500 coflows and 20 maximum coflow width were due to performance constraints.

We compared the performance of our Sincronia implementation against an existing Coflow scheduling solver, 'CoflowSim' [4]. According to the project `README`, the average CCT minimizing algorithm 'Varys' [5] is the implementation for CoflowSim. In the Sincronia paper by Agarwal, Varys was used as an existing CCT minimizing algorithm for comparison. We attempt to perform a similar comparison by using CoflowSim as the frontend to Varys.

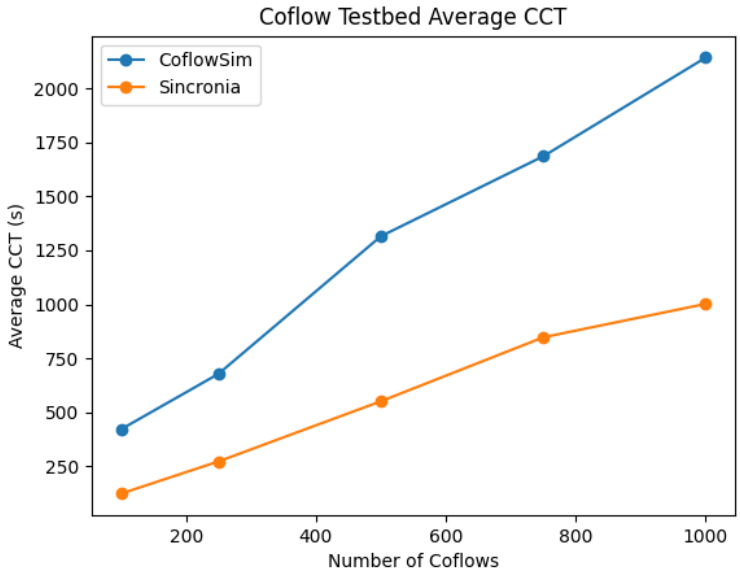
Our Sincronia implementation has an ingress & egress port bandwidth of one megabyte per second (1MBps), and instantaneous scheduling as soon

as coflows are available. Thus we configured CoflowSim to match these conditions as closely as possible by adjusting the parameters in `coflowsim/src/main/java/coflowsim/utils/Constants.java`; `RACK_BITS_PER_SEC=1024*8192`, and `SIMULATION_SECOND_MILLIS=8` respectively. The workload generator source code was also modified to align the access link bandwidth to represent 1MBps per port; `ACCESS_LINK_BANDWIDTH=2*NUM_INP_PORTS`.

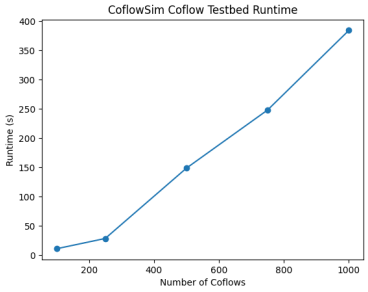
We used the command-line argument `FIFO` as the scheduling algorithm for CoflowSim in this experiment, although there is no significant reason for this choice beyond being a clairvoyant scheduler. We reached out to Agarwal inquiring about exactly how they tested Sincronia against Varys but did not receive a response. In order to use the workloads generated by the Sincronia workload generator, we used a script to convert the trace format so that CoflowSim could accept it.

The experiment was initially conducted on the University of Waterloo's Math Faculty Computing Facility (MFCF) student Linux machines, and retimed on a computer running Ubuntu 20.04 with a Quad Core i5 2400 3.1 GHz processor and 4 GB of RAM.

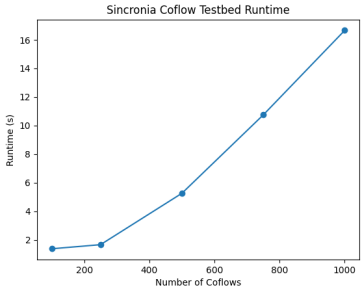
3.3. Experiment Results



(a) Coflow testbed average CCT comparison.

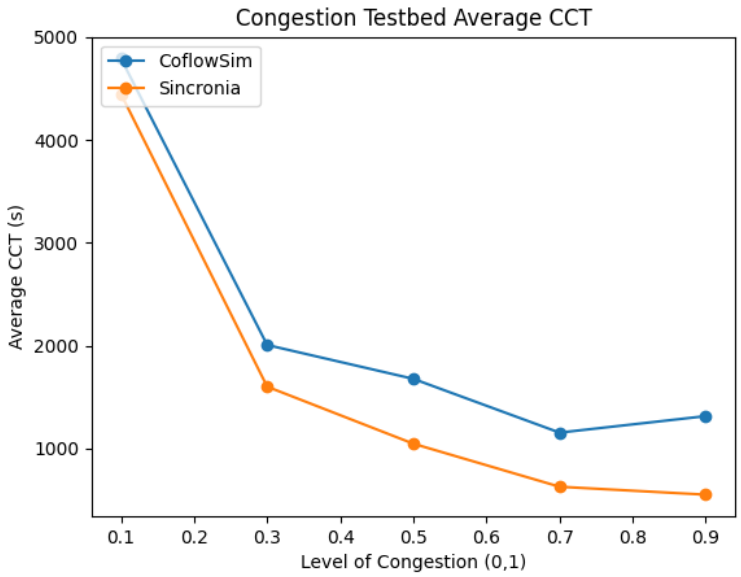


(b) CoflowSim Coflow testbed runtimes.

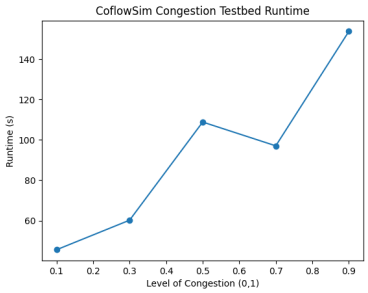


(c) Sincronia Coflow testbed runtimes.

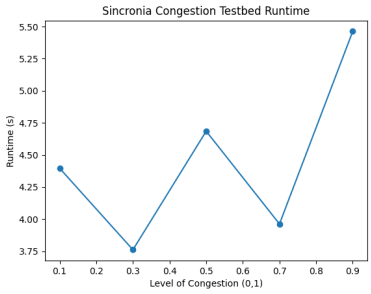
Figure 3: Testbed for variable number of coflows (0.9 load, 0.5 contention, 20 max coflow width, 150 ports)



(a) Congestion testbed average CCT comparison.

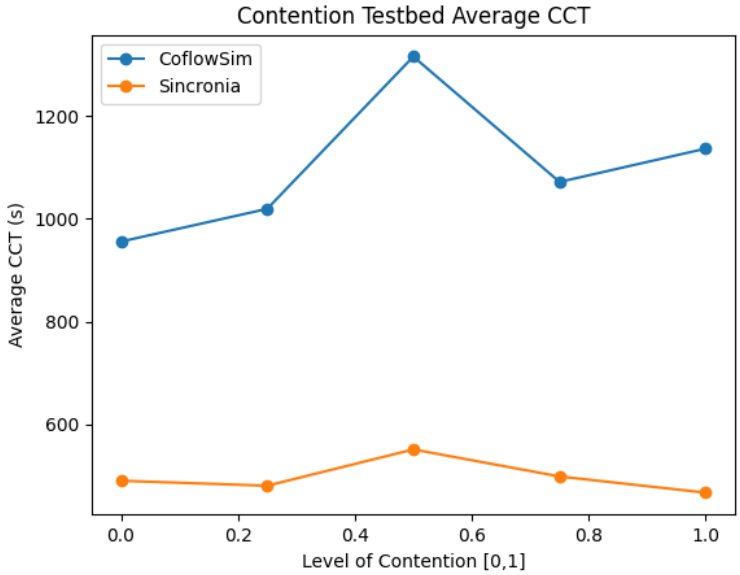


(b) CoflowSim Congestion testbed runtimes.



(c) Sincronia Congestion testbed runtimes.

Figure 4: Testbed for variable level of network congestion (500 coflows, 0.5 contention, 20 max coflow width, 150 ports)



(a) Contention testbed average CCT comparison.

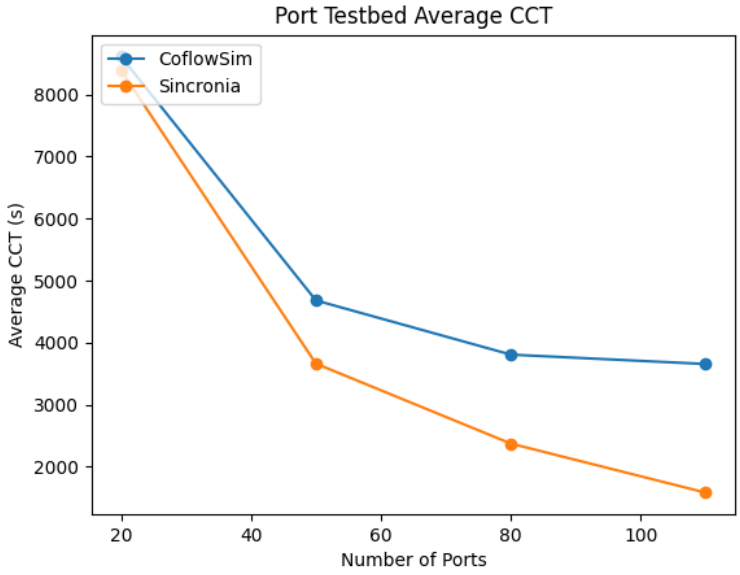
figures/contestion_testbed_runtime.coflowsim.png

(b) CoflowSim Contention testbed runtimes.

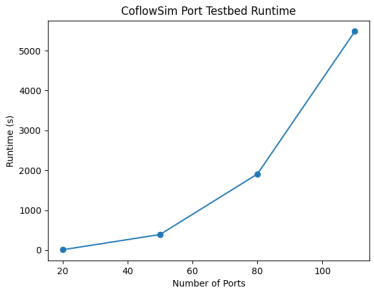
figures/contestion_testbed_runtime.sincronia.png

(c) Sincronia Contention testbed runtimes.

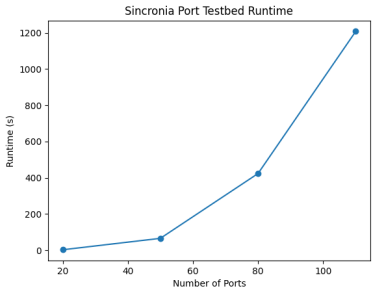
Figure 5: Testbed for variable level of port contention (500 coflows, 0.9 load, 20 max coflow width, 150 ports)



(a) Port testbed average CCT comparison.



(b) CoflowSim Port testbed runtimes.



(c) Sincronia Port testbed runtimes.

Figure 6: Testbed for variable number of ports (500 coflows, 0.9 load, 0.5 contention, max coflow width = number of ports

4. Analysis

The following analysis come with these disclaimers:

- We acknowledge the small dataset and use of singular data points
- We acknowledge the likelihood for mis-implementation of Sincronia
- We acknowledge the likelihood of mis-configuration of the Sincronia workload generator

– We acknowledge the likelihood of mis-configuration of CoflowSim

– The runtimes of CoflowSim are significantly inflated due to the miniscule timestep used in configuring CoflowSim. This results in an extremely large amount of idle timesteps.

4.1. Coflow Testbed

The results in Figure 3 (a) do not provide much information due to the lack of data points. The limited compute resources in the experiment prevented tests for larger quantities of coflows, so it is possible that the trends of both CoflowSim and Sincronia are non-linear. Thus the only conclusion to be drawn from Figure 3 (a) is that average CCT trends upwards as the number of coflows increases, and that Sincronia yields a significantly lower (approx. 1/2) average CCT compared to the experiment's configuration of CoflowSim.

Figure 3 (b) and (c) are similarly limited due to the lack of data points, so we can only conclude that the runtime trends upward as the number of coflows increases.

4.2. Congestion Testbed

The results in Figure 4 (a) imply that the performance of both algorithms improves as the network congestion increases. However, the reason for this trend is in the workload generator; high network congestion yields more dense arrival times. Hence for a fixed number of coflows, high network congestion results in lower arrival times. Thus the results from this graph are useless for analysis.

Figure 4 (b) and (c) are limited due to the lack of data points, but it appears that the runtime trends slightly upwards as congestion increases for CoflowSim. A relationship for Sincronia is unclear due to the lack of data points.

4.3. Contention Testbed

The results in Figure 5 (a) imply that the performance of both algorithms are for the most part unaffected by port contention. There is a spike at 50 percent contention, though this is likely just due to fluctuations between individual workloads and would require more data points to verify.

Figure 5 (b) and (c) are limited due to the lack of data points, but it appears that the runtime trends slightly upwards as contention increases for CoflowSim. A relationship for Sincronia is unclear due to the lack of data points.

4.4. Ingress & Egress Port Testbed

The results in Figure 6 (a) seem to indicate that both algorithms' average CCT trends asymptotically towards some minimum value as the number of ports increases. This makes sense as we are keeping other variables (namely the number of coflows) fixed. This could verified through additional data points, however we were again limited by our compute resources.

Figure 6 (b) and (c) seem to indicate that the processing difficulty increases non-linearly as the number of ports per coflow increase. This makes sense, and again could be verified with additional data points.

5. Conclusion

Utilizing a small testbed, we are able to observe that Sincronia performs at the level of existing coflow completion time minimizing algorithms. Furthermore, the claim that Sincronia improves approximately 1.7 times over Varys at minimizing average coflow completion time holds true over our testbed.

5.1. Future Directions

In our implementation of Sincronia, we allow for non-unit weights for coflows. However this capability was not tested. If CoflowSim (Varys) or another existing coflow completion time minimizing algorithm supports non-unit weights, we could run the same testbed but with weighted coflows and compare how that affects the performance.

Sincronia section 5.2 claims that in the offline case, it achieves an average of 1.7 times improvement over Varys. Assuming our experiment was a fair comparison between Sincronia and Varys, this claim appears to have merit for the online (non-zero arrival times) case as well. We could more specifically target this claim by creating offline (zero arrival times) workloads.

Sincronia section 5.2 also claims in the online case an average 'slowdown' of 1.7 times at 0.9 load for generated workloads with 4000 coflows. The 'slowdown' metric is described as the factor by which a given Coflow's CCT is increased with respect to if it had arrived at an unloaded network. We could examine this claim as well by examining the 'slowdown' factor over various workloads.

Furthermore, Sincronia section 5.2 claims in the 'Sensitivity Analysis' that performance improves as network load decreases from 0.9 to 0.7. This too could be verified using the same 'slowdown' metric with workloads at various levels of network congestion.

References

[1] A. N. R. A. D. S. Saksham Agarwal, Shijin Rajakrishnan, A. Vahdat, Sincronia: Near-optimal network design for coflows, SIGCOMM '18: ACM SIGCOMM 2018 Conference (2018). doi:https://doi.org/10.1145/3230543.3230569.

[2] S. Agarwal, Coflow workload generator, https://github.com/sincronia-coflow/workload-generator, 2018.

[3] Coflow-benchmark, https://github.com/coflow/coflowsim, 2015.

[4] M. Chowdhury, A. Jajoo, Coflowsim, https://github.com/coflow/coflow-benchmark, 2017.

[5] M. Chowdhury, Varys, https://github.com/coflow/varys, 2015.