A Case for Performance-Centric Network Allocation

Gautam Kumar, Mosharaf Chowdhury, Sylvia Ratnasamy, Ion Stoica
University of California, Berkeley
{*gautamk, mosharaf, sylvia, istoica*}*@eecs.berkeley.edu*

Abstract

We consider the problem of allocating network resources across applications in a private cluster running data-parallel frameworks. Our primary observation is that these applications have different communication requirements and thus require different support from the network to effectively parallelize. We argue that network resources should be shared in a performance-centric fashion that aids parallelism and allows developers to reason about the overall performance of their applications. This paper tries to address the question of whether/how fairness-centric proposals relate to a performance-centric approach for different communication patterns common in these frameworks and engages in a quest for a unified mechanism to share the network in such settings.

1 Introduction

The question of how to allocate resources – CPU, memory, and network bandwidth across jobs is central in the operation of datacenters. Much of the discussion on resource allocation, particularly in the context of network bandwidth, has centered around the principle of fairness [4, 6]. Fairness is an appealing target since (done right) it can offer many desirable properties including strategy-proofness and non-starvation. However, the drawback of a purely fairness-centric approach is that it may offer users/developers little guidance on the performance they can expect while scaling their applications. The high level question this paper aims to raise is whether we can allocate datacenter resources in a manner that allows developers to reason about the performance they can expect and whether/how a performance-centric approach relates to fairness-centric proposals. We believe that understanding the options and trade-offs here would be particularly valuable to Internet datacenters operated by, and in the interests of, a single organization – e.g., datacenters at Facebook, Microsoft or Google, where concerns over competitive or non-cooperative user behavior are less severe (though, admittedly, not non-existent!) than in general cloud environments.

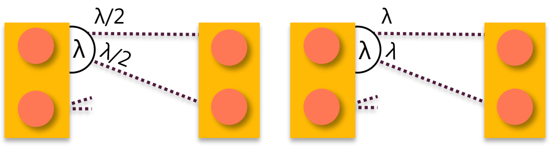
In this paper, we focus on performance as measured by job completion time and on applications that embrace *data parallelism* – by which we mean that an application can partition its input into multiple sets and operate on individual sets in parallel. That is, for a fixed input the total computation remains the same whether it is done on a single machine or it is partitioned into N sets and done across N machines in parallel. These applications appear amenable to a simple, intuitive model for relating resource allocation to performance, namely: given N times more resources, such applications can expect to complete N times faster.[1] We show that while resources such as CPU and memory adhere to this simple model under data parallelism, the case of the network is more complicated. This is illustrated in the discussion that follows.

Frameworks such as MapReduce [3], Dryad [5], and Spark [8] enable many applications to exploit such data parallelism and typically proceed in several *computation* stages requiring *communication* between them. Performance of data-parallel applications depends on a variety of factors like CPU, memory, cache contention, disk contention etc.; however, in this paper, we focus only on the network communication aspect of these frameworks. Our primary observation is that, while the total computation in such frameworks is invariant to the number of machines that are used (thus adhering to the

[1]We will not get exactly N speed-up in practice due to limited resources but it is a useful rule of thumb.

above mapping between scaling resources and performance), we find that the total communication (i.e., the total intermediate data to be transferred) between stages may not follow this simple mapping. That is, data parallelism need not imply network parallelism.

Existing network allocation mechanisms fall short because either they only make first-order approximation of proportionality (e.g., proportional to the number of VMs [6] or the number of sources [7]) in the absence of application semantics, or they require applications to explicitly specify their network requirements [1] which are often not known. We observe that the extent to which the simple model of parallelism breaks down actually depends on the communication semantics of cluster computing applications. We illustrate this by picking two types of transfers [2] that lie at the opposite ends of the spectrum (we later generalize this notion in §2.3):



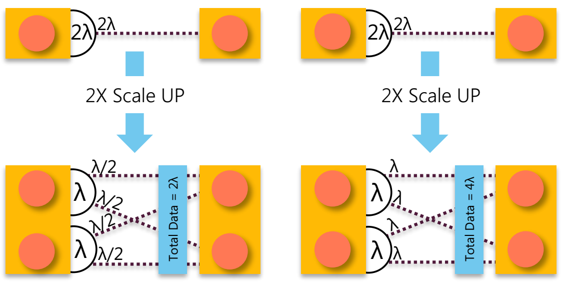
(a) Shuffle (b) Broadcast

Figure 1: Different types of transfers.

- **Shuffle**: Each node in the previous stage partitions its computed data into N sets and sends each set to one of the nodes in the next stage (e.g., the λ data generated in Figure 1(a) is split in two $\frac{\lambda}{2}$ sets and sent to the two nodes on the right).
- **Broadcast**: Each node in the previous stage sends all of its computed data to every node in the next stage (e.g., all of the λ data generated in Figure 1(b) is sent to each of the nodes on the right).

Different applications require different types of transfers. Applications suited for the MapReduce framework require a shuffle of the intermediate data, while several machine learning applications require broadcast [9].

Let us now understand why the invariant on the total communication does not hold with the help of a specific example. Consider Figure 2, where a job wants to split its computation across two machines (instead of just one) for both the stages and understand the overhead of parallelization on the communication. Assume that when the job ran with only one machine in both the stages, it generated 2λ amount of data to be transferred to the second stage. We consider both the cases, one where the application required a shuffle of the intermediate data and the other where the application required broadcasting the intermediate data. While in the shuffle case, the total communication remains the same (2λ), for broadcast the total communication doubles to 4λ. Hence, we note that in order to preserve the notion of parallelism, while it is sufficient to give *twice* as much network bandwidth in the shuffle case (a linear scaling), we need to give *four times* as much network bandwidth in the broadcast case (a quadratic scaling). It is important to note that this overhead in communication is not a result of a breakdown in an application's parallelism; i.e., the application did not suddenly generate more data (which remains at 2λ) but rather that the extra network traffic was a result of scaling up. In short: scaling even a simple application can result in an additional network overhead (depending on the transfer type required by the application). We show that accounting for this potential overhead is key to achieving a consistent model that maps resource allocation to job completion times.



(a) Shuffle (b) Broadcast

Figure 2: Total communication of a job scales differently depending on the type of transfer it uses.

Building on the above observations, we propose a new perspective on network sharing in which network resources are allocated in support of preserving the intuitive benefits of parallelism. This is in contrast to prior proposals that argue for network allocations that are proportional to the number of nodes that a job uses (e.g., FairCloud [6]) or proportional to the number of flows that a job instantiates (e.g., per-flow sharing as approximated by TCP). In support of this perspective, we aim for a performance-centric sharing mechanism that isolates the achievable speed-up due to parallelism and the performance degradation due to limited resources ensuring that, *in the case where the network is the bottle-*

neck, jobs are penalized equally w.r.t their completion times; i.e., all else being equal, jobs suffer an equal degradation in their job completion times. We then make the following observations:

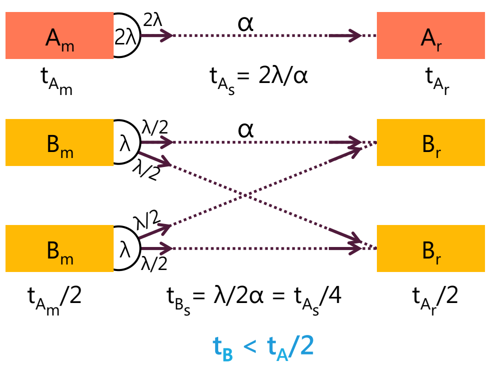
- For clusters running applications requiring *only* shuffles (e.g., MapReduce), sharing the network in proportion to the number of machines that a job uses [6] is the performance-centric sharing mechanism and that per-flow sharing (equal share for every flow) hurts the performance for small jobs.
- For clusters running applications requiring *only* broadcasts (e.g., several machine learning applications), per-flow sharing (approximated by TCP) results in preserving the benefit of parallelism in completion times and proportional sharing (in terms of the number of machines) limits parallelism for large jobs.
- For frameworks supporting multiple types of transfers, the performance-centric sharing mechanism assigns network shares based on the semantics of the transfer.

2 Network Sharing to Support Parallelism

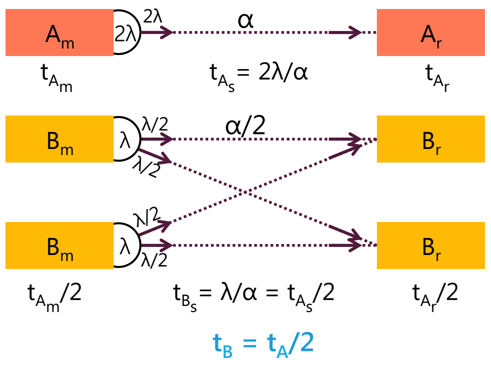
In this section, we first consider two types of data-parallel clusters, one supporting applications requiring only shuffles and the other supporting applications requiring only broadcasts, and we show that from the perspective of parallelism, different network sharing mechanisms are required in the two clusters. We qualify these transfers in terms of their behavior when the application scales and generalize them to arbitrary types. The perspective of parallelism that we take in this context is that a scale-up of N should give the job a speed-up of N for a fixed *input* size.

2.1 Shuffle

Let us take a MapReduce cluster that runs jobs requiring a shuffle of the intermediate data between the two computation stages (map and reduce). Consider a job that needs to perform word count across two files. For simplicity, assume that the two files represent equal amount of work to do and the key-space is uniformly partitioned. We study the following two alternatives as shown in Figure 3.



(a) Unbalanced network share (shuffle)

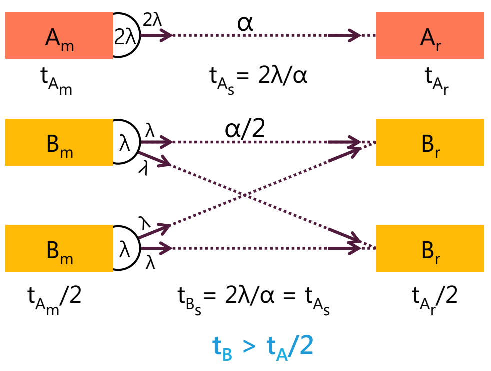


(b) Balanced network share (shuffle)

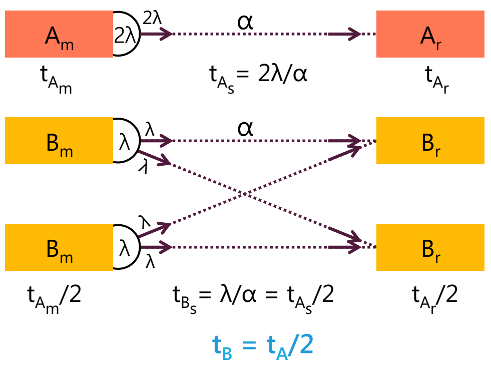
Figure 3: Network allocation of a MapReduce job.

- A: A mapper sequentially processes the two files, transfers 2λ data over the network (assuming that a single file generates λ data), and then a single reducer processes this 2λ data.
- B: Two mappers process the files in parallel and each mapper generates λ data, transfers $\frac{\lambda}{2}$ to each of the reducers over the network, and finally each reducer processes λ data ($\frac{\lambda}{2}$ from each mapper) in parallel.

We denote the times for the three phases (map, shuffle, and reduce) as t_m, t_s, and t_r; the total time of the job being $t_m + t_s + t_r$. With the above assumption of uniformity, $t_{B_m} = \frac{t_{A_m}}{2}$ and $t_{B_r} = \frac{t_{A_r}}{2}$ since they were done in parallel. Let us now consider the shuffle phase, a per-flow mechanism would give an equal share (α) to each of the flow. This implies that $t_{B_s} = \frac{\lambda}{2\alpha} = \frac{\frac{2\lambda}{\alpha}}{4} = \frac{t_{A_s}}{4}$, which implies $t_B < t_A/2$. Hence, with per-flow sharing, the job achieves a larger share of the network that it needs in order to achieve a speed-up of 2. Since the total network resources are finite, this comes at the cost of taking away the network share from other jobs which will now get a smaller share than what they should. In other words, a per-flow sharing mechanism gives larger



(a) Unbalanced network share (broadcast)



(b) Balanced network share (broadcast)

Figure 4: Network allocation of a broadcast-only job.

jobs more than their required share and thus hurts performance for the small jobs. Instead, if the rate of each of the flows was reduced to $\frac{\alpha}{2}$, the new time for shuffle will become $t_{B_s} = \frac{\lambda}{\alpha} = t_{A_s}/2$ making $t_B = t_A/2$. The total network share of the job becomes equal to 2α, increasing in *proportion* with the the number of machines and thus proportional mechanisms (e.g., [6]) are performance-centric in this case.

2.2 Broadcast

Let us now consider another cluster that supports applications that *only* broadcast the intermediate data from one computation stage to another. While several machine learning applications need this model (e.g., eigenvalue decompositions or the alternating least squares method [9] used for the Netflix prize winning entry that partitions the feature vectors for users and movies separately and requires broadcasts of the vectors between the two stages in an iterative fashion), the illustrative example we look at is the problem of determining whether a musical piece has been composed by Bach or Britney Spears, and we have a collection of their musical pieces to train. The job first needs to compute feature vectors over the two sets of input data (one for pieces of Bach and the other for Britney), and then both detectors are trained using the feature vectors corresponding to both Bach's and Britney's pieces thus necessitating a broadcast of the intermediate data to train the detectors in parallel. Similar to the shuffle case, we consider the following two alternatives for the job to run (Figure 4).

- `A`: In the first stage, training data corresponding to both categories is processed sequentially and the computed feature vectors (a data of 2λ assuming the training data consisted of same number of images for both the categories) are transferred to the next stage where both the classifiers (both of them requiring the entire 2λ data since they require both positive and negative examples) are trained sequentially.

- `B`: The feature vector computation for both the categories is done in parallel in the first stage, one node corresponding to pieces by Bach and the other corresponding to pieces by Britney Spears. Each of them generate λ amount of data (the feature vectors) and need to pass the *entire* data to *each* of the two classifiers in the next stage.

We use a notation similar to the shuffle case, but rename transfer time as t_b. Again assuming uniformity of the input data, $t_{B_m} = \frac{t_{A_m}}{2}$ and $t_{B_r} = \frac{t_{A_r}}{2}$ since they were done in parallel. For the transfer, proportional allocation, that was desirable for shuffles, would give $\frac{\alpha}{2}$ to each of the flows of `B`. This implies that $t_{B_b} = \frac{2\lambda}{\alpha} = t_{A_b}$ and thus $t_B > t_A/2$. Therefore, proportional network allocations would prohibit achieving ideal speed up and thus limit parallelism for large jobs for broadcast-only clusters. Instead if each flow of `B` was still given a share of α (per-flow), the new time for the transfer will become $t_{B_s} = \frac{\lambda}{\alpha} = t_{A_s}/2$ making $t_B = t_A/2$. Therefore, from the perspective of parallelism, per-flow allocations are performance-centric in this case.

To summarize, we argue that parallelism-driven network sharing yields different design points for shuffle-only and broadcast-only clusters. Performance-centric network allocations, therefore, accommodate the semantics of the transfer. We present a concise qualitative diagram to illustrate this notion in Figure 5. The x-axis denotes the degree of parallelism, i.e., the scale up in terms of the number of machines and the y-axis denotes the potential speed up. For shuffle, per-flow based net-

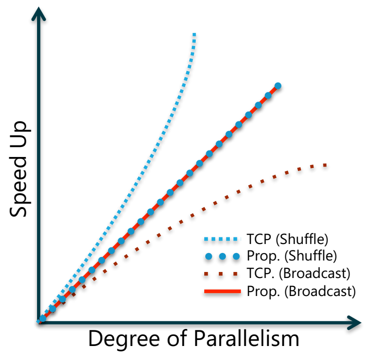


Figure 5: Network sharing mechanisms should take the semantics of transfers into account.

work sharing (e.g., TCP) gives higher speedup than the computational speed-up, implying that jobs get more than their required share, while proportional allocations achieve the required balance. However, proportional allocations in the case of a broadcast makes the jobs perform worse compared to their expected speed-up and in this case, per-flow allocations achieve the balance.

2.3 Complexity of a Transfer

In the previous two subsections we observed that a scale up of N does not affect the total amount of data transferred during shuffles, but it makes it go up by a factor of N for broadcasts. We use this observation to generalize the notion of a transfer. We call a transfer to be an x_N-transfer if x is the factor by which the amount of data transferred increases when a scale up of N is done, $x \in [1,N]$.[2] A shuffle is a 1_N-transfer and a broadcast is an N_N-transfer. Performance-centric network allocation in a cluster that runs applications requiring *only* x_N-transfers means the following: when the applications are scaled up by a factor of N, their network shares should be increased by a factor of $(N \times x)$ to achieve a (proportional) speed-up of N.

3 Heterogeneous Frameworks and Congested Resources

In the previous section, we considered frameworks that exclusively used only shuffle, broadcast or the more general x_N-transfer for communicating between different stages. We now address the question of how to share the network in frameworks that use more than one of the above transfers and understand the behavior when resources get bottlenecked. As an example, frameworks like Spark [8] and Dryad [5] support both the shuffle and the broadcast primitives. We showed that for shuffles, the network share should be proportional to the number of machines that the job uses, say N. For broadcasts, this share should be proportional to N^2.

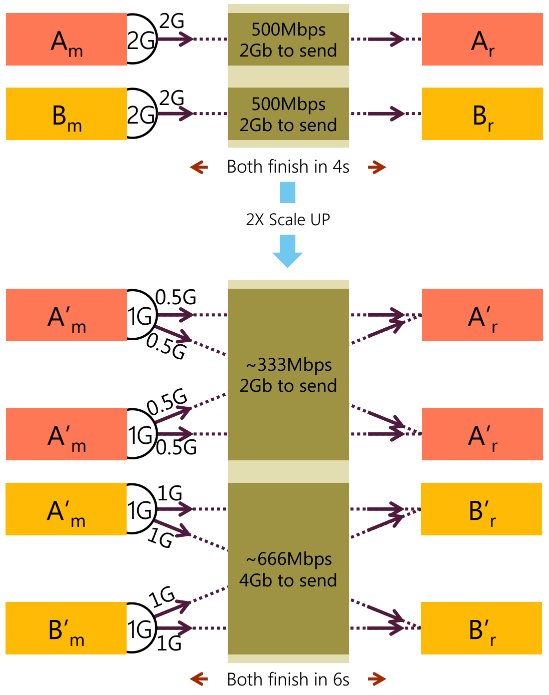


Figure 6: Job completion times degrade uniformly when resources are bottlenecked.

We argue that even with different types of transfers running in a cluster, the notion of performance-centric allocations that implies equal degradation in job completion times when the resources get bottlenecked can be retained if independent decisions are made depending on the type of the transfer. In essence, the idea is to effectively isolate the speed-up achievable due to parallelism and the degradation due to limited resources. Thus, in the event of contention, the job's new completion time, y', can be compared to its original running time, y, by the following equation:

$$y' \leftarrow (\alpha) \times \left(\frac{y}{N}\right)$$

where α, the degradation due to congestion, is the same for the jobs and thus, the completion time de-

[2] $x > N$ implies that the application would be generating more data than before; this is not possible because scaling up of a data-parallel application does not change its input, nor does it change the computation function. $x \geq 1$ is a trivial lower bound assuming that the application will generate at least as much data as it was generating before the scale up.

grades uniformly for the both of them. We illustrate this with the help of an example. Consider the word count job, A, and the musical-piece categorization job, B, running together in a cluster as illustrated in Figure 6. For simplicity, we assume that all transfers take place on a single link (with capacity $C = 1$Gbps) and there is no other workload in the cluster. If both the jobs used a single machine in both the stages, assuming that they had same amount of data (2Gb) to transfer, the transfer in both the cases finishes in 4s since both of them get a share of $\frac{C}{2} = 500$Mbps. Now when both the applications scale to use two machines on either side, the share of job A is proportional to $2^1 = 2$ and the share of job B is proportional to $2^2 = 4$. Therefore, A gets $\frac{C}{3} \approx 333$Mbps and B gets $\frac{2C}{3} \approx 666$Mbps implying that both the transfers finish in 6s and thus face an equal degradation.

4 Discussion and Future Directions

There has been a lot of recent work on how to share the network in a datacenter. The general focus of these efforts has been to support multi-tenancy and ensure isolation and fairness [7, 1, 6] between different tenants where properties like strategy-proofness, proportionality etc. are desirable. In this paper, instead of talking about the problem of network sharing in general settings, we focus on private clusters running data-parallel frameworks. This setting provides us with an alternate view of the world, where the entities are not malicious and the aim is to achieve parallelism for the workloads. We present a new perspective to share the network which considers whether the network provides the desired support for the applications to effectively parallelize. We observe that the present approaches do not generalize to achieve this goal and to remedy it, network sharing should be done based on the application semantics. In particular, we showed why the sharing mechanism should be different for frameworks shuffling the intermediate data and those employing broadcast as the communication pattern. We also gave some intuition about network sharing when the framework provides both (along with other) communication primitives, though this should only be treated as a starting point.

In future work, we want to analytically and experimentally understand how these frameworks behave when the ideas presented in this paper are used. In particular, we want to understand whether mechanisms based on this criterion only result in redistribution of the completion times across different jobs or are they also able to achieve a balance to reduce the total completion time of all jobs. One other question that remains to be answered is what happens when the scaling up factors are different on either side; the analysis in that case becomes much more complicated and therefore requires further thought.

Acknowledgments

We are grateful to the anonymous reviewers, Lucian Popa, Bharath Hariharan, Aurojit Panda and the members of the AMPLab for their helpful comments. This research was supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

References

[1] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards Predictable Datacenter Networks. In *ACM SIGCOMM, 2011.*

[2] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM, 2011.*

[3] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI, 2004.*

[4] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI, 2011.*

[5] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM EuroSys, 2007.*

[6] Popa, L., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. FairCloud: Sharing the Network in Cloud Computing. In *ACM HotNets, 2011.*

[7] Shieh, A., Kandula, S., Greenberg, A., Kim, C., and Saha, B. Sharing the Data Center Network. In *USENIX NSDI, 2011.*

[8] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI, 2012.*

[9] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM, 2008.*