Distributed Coded Computation

Pedro J. Soto

March 2021

Abstract

A ubiquitous problem in computer science research is the optimization of computation on large data sets. Such computations are usually too large to be preformed on one machine and therefore the task needs to be distributed amongst a network of machines. However, a common problem within distributed computing is the mitigation of delays caused by faulty machines in the network or traffic congestion. This can be performed by the use of coding theory to optimize the amount of redundancy needed to handle such faults. This problem differs from classical coding theory insofar that it is concerned with the dynamic coded computation on data rather than just statically coding data without any consideration of the algorithms to be performed on said data. Of particular interest is the operation of matrix multiplication, which happens to be a fundamental operation in many big data/machine learning algorithms, which is the main focus of this paper. Two wonderful consequences of the (bi-)linear nature of matrix multiplication is that it is both highly parrallelizable and that linear codes can be applied to it; making the coding theory approach a fruitful avenue of research for this particular optimization of distributed computing.

Contents

Chapter 1

Introduction

As the rate at which CPU's get smaller slows (since there is an atomic limit on their size), distributed computing has become a more ubiquitous and necessary topic in both research and the lives of ordinary consumers of computer technology. Another reason for the rise of distributed computing is that data sets have become very large and the computations performed on them must be split amongst many machines. In a distributed setting a large task is split up into smaller tasks that are run in parallel amongst the machines, or *nodes*, in the network.

However, it is well known that nodes in a distributed infrastructure are commonly composed of commodity hardware which are more likely to be subject to various faulty behaviors [HGZ$^+$17]. One example of such a fault is when a node may experiences a temporary performance degradation due to load imbalance or resource congestion [LLP$^+$18]. In particular, the performance of virtual machines in Amazon EC2 clusters have been observed to have a performance degradation of up to a factor of 5 [TLDK17, LLP$^+$18]. A node may even fail to complete a task due to hardware failures, network partition, or power failures. In a Facebook data center, it has been reported that up to more than 100 such failures can happen on a daily basis [RSG$^+$13, SAP$^+$13]. Therefore, when the computation is distributed onto multiple nodes, its progress can be significantly affected by the tasks running on such slow or failed nodes, which we call *stragglers*.

1.1 Problem Formulation

In general we have some computation f that we wish to perform on some data-set D. If the data-set D or the computation f becomes too large/expensive to perform on one machine, then the computation and the data set must be split up, or *partitioned*, into smaller *tasks* $\mathcal{P} : f(D) \mapsto f_1(D_1), ..., f_k(D_k)$ as illustrated in fig 1.1. We wish to do so in a way that recreating the original job, $f(D)$ from the tasks $f_1(D_1), ..., f_k(D_K)$ has very little overhead (*i.e.,* we wish

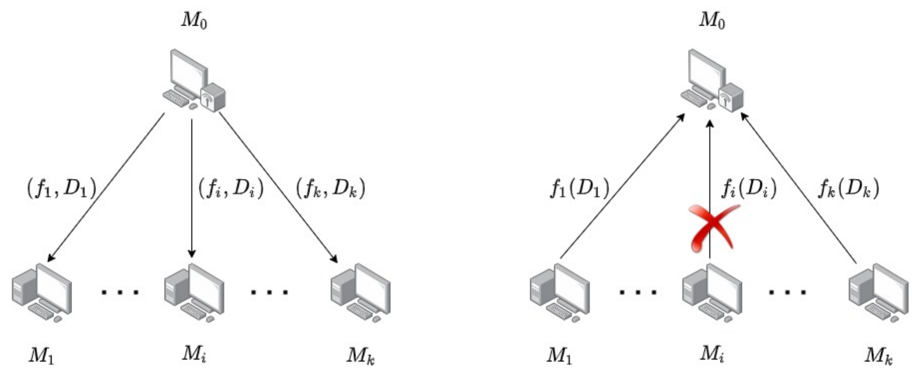

Figure 1.1: The task $f(D)$ is too large to do on machine M_0 so it partitions the job into some smaller tasks $f_1(D_i), ..., f_k(D_k)$ and distributes it amongst the workers. Supposing that the machine M_i takes a long (possibly infinite) time t_i to finish then the desired computation of $f(D)$ is delayed by time t_i.

to minimize the complexity of $\mathcal{P}^{-1}$). **In particular, if one of the machines, say M_i, takes a long time, say t_i, to compute $f_i(D_i)$ then the entire algorithm is delayed by the time t_i.** In the worst case, machine M_i never finishes and the computation $f(D)$ is never completed. The main idea is to add some redundant tasks $f_{k+1}(D_{k+1}), ..., f_{k+r}(D_{k+r})$ to $\mathcal{P}$ so as to mitigate the possibility of stragglers (*i.e.,* so that we can recreate $f(D)$ without needing the data $f_i(M_i)$). However, this cannot be done naively; we will show in sec. 1.2 that merely replicating the same tasks on other workers will not suffice since it is highly inefficient and can suffer from the same faults it is trying to solve. Therefore it becomes necessary to apply error-correcting codes, or more precisely *erasure codes*, to the problem. In particular the naive partition, replication, and recreating in $\mathcal{P}, \mathcal{P}^{-1}$ are replaced with *encoding* and *decoding* functions $\mathcal{E}, \mathcal{D}$ of an erasure code.

1.2 Motivating Example

Consider the following toy example: M_0 wishes to compute the product of the two matrices

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}, \ B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

but computing AB would take too long to compute on one machine. M_0 could send $A_1 = \begin{bmatrix} a_{1,1} & a_{1,2} \end{bmatrix}$ to M_1 and send $A_2 = \begin{bmatrix} a_{2,1} & a_{2,2} \end{bmatrix}$ to M_2 and in addition send B to both workers as indicated by fig. 1.2. Since it is easy to verify that

$$\begin{bmatrix} A_1 B \\ A_2 B \end{bmatrix} = \begin{bmatrix} A_1 B \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ A_2 B \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ 0 & 0 \end{bmatrix} B + \begin{bmatrix} 0 & 0 \\ a_{2,1} & a_{2,2} \end{bmatrix} B = AB, \quad (1.1)$$

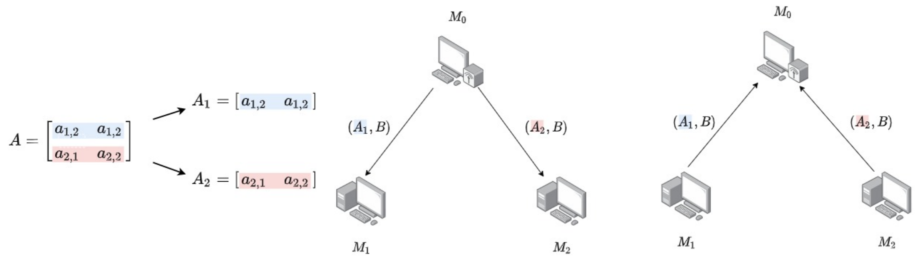

Figure 1.2: The master first splits A into smaller sub-matrices A_1 and A_2. Then the master sends the tasks (A_1, B) and (A_2, B) to be multiplied by machines M_1 and M_2 respectively. As indicated by eq. (1.1) the master can then recreate the original job AB from these two sub-tasks.

a valid scheme would be to have M_1 compute A_1B and have M_2 compute A_2B and then combine the results in the obvious way indicated by eq. (1.1) as indicated in fig. 1.2.

A naive way to mitigate the possibility of M_1 or M_2 becoming a straggler is to use *replication*. For example we can replicate the tasks on two other machines M_3, M_4 by sending (A_1, B) to M_3 and sending (A_2, B) to M_4.

1.3 Outline of the Survey

Chapter 2

Background

2.1 Distributed Systems

2.2 Coding Theory

2.3 Linear Algebra and Learning Algorithms

Chapter 3

Variations of the Main Problem (and Their Proposed Solutions)

3.1 One Large Matrix Multiplication

3.2 Multiple (*i.e., Batch*) Matrix Multiplications

3.3 Partial Results/Partial Stragglers

3.4 More Complicated Functions/Machine Learning Applications

Chapter 4

Our Work

4.1 Dual Entangled

4.2 Spinner

4.3 Rook Poly

Chapter 5

Future Work

Bibliography

[HGZ+17] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2017.

[LLP+18] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2018.

[RSG+13] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[SAP+13] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.

[TLDK17] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *International Conference on Machine Learning (ICML)*, pages 3368–3376, 2017.