Using Many Machines to Handle an Enormous Error-Correcting Code

Jon Feldman

Google, Inc.
1440 Broadway, 21st Floor.
New York, NY, 10025
jonfeld@google.com

Abstract— **We investigate the problem of using many machines to represent, encode and decode an error-correcting code with an extremely large block length. Standard algorithms for encoding and decoding run into problems when scaled to a block length that does not allow random access to the data. We apply the massive computing infrastructure at Google together with the MapReduce programming abstraction to encode and decode a Tornado code over the erasure channel.**

I. Introduction

New advances in error-correcting codes are making it possible to achieve nearly optimal tradeoffs between rate and error-correcting performance, with simple and efficient encoding and decoding algorithms. For example, Tornado codes [1] offer essentially optimal performance in the erasure channel, with linear-time encoding and decoding algorithms. Guruswami and Indyk [2] give a class of linear-time encodable and decodable near-MDS codes over channels with errors. Turbo codes [3] are practical codes that operate over noisy channels, and achieve superb performance with very simple encoding and decoding schemes.

When these schemes are scaled up to extremely large block lengths, however, new challenges arise. Suppose we had a very large data file—say a terabyte (2^{40} bytes)—that we need to store on some faulty medium. We would like to encode this file using an error-correcting code without breaking it up into pieces, thus taking advantage of its large size for better error-correcting performance. At this size, it is impossible even to write down a generator matrix, let alone use it for encoding. Thus an LDPC code [4] or a Reed-Solomon code [5] is basically out of the question. Even linear-time schemes like Tornado codes suffer the problem of needing fast random access to information or code bits, which at this size is quite inefficient; indeed a data file of this size cannot fit into the memory (or even the disk) of a commodity machine by today's standards.

For extremely large block lengths, we need encoding and decoding algorithms that use only a few "passes" over the data, and preferably each pass can be performed in parallel using multiple machines. Such large-scale operations require a computing framework that can handle massively parallel algorithms, dealing with issues like fault-tolerance, task scheduling and inter-machine communication.

Google's computing infrastructure, coupled with their implementation of the MapReduce programming model [6] provides such framework. In this preliminary report, we explore the application of this infrastructure to the problem of encoding and decoding a very large Tornado code. We give some background on both Tornado codes and MapReduce, then go into detail about how encoding and decoding are performed. Our experimental work is ongoing, and will be summarized in a subsequent report.

II. Background

A. MapReduce

MapReduce is a powerful programming abstraction for performing operations over large data sets[6]. The user provides a *map* function and a *reduce* function; the map function processes the input, producing a set of (*key*, *value*) pairs that are passed to the reduce function grouped by key. These functions are executed in parallel (as appropriate) over the large data set, hiding details like load-balancing, fault-tolerance, and inter-process communication. The implementation of MapReduce at Google can routinely handle many terabytes of data at one time, running over thousands of commodity machines. The simple interface allows the user to define new processes efficiently, and automates all the complexities of parallelization and scaling that arise when handling large amounts of data over many machines.

For the purposes of running a MapReduce, we regard our input data file as a set of records. The user-defined *map* function takes as input one of these records, and outputs a set of (*key*, *value*) pairs. (The form and purpose of these pairs is up to the user.) The user-defined *reduce* function takes as input a key k and *set* V of values, and produces arbitrary output. When the user executes the MapReduce, the *map* function is executed on every record, producing a large set of these (*key*, *value*) pairs (repeats are allowed). For each *key* k that is output by the *map* phase, all the values from pairs with key k are gathered into a set V (this is handled automatically by the MapReduce library), and the *reduce* function is run on (k, V), producing whatever output the user defines. For more details on how these phases are implemented over multiple machines, we refer the reader to [6].

1-4244-0036-8/06/$20.00 ©2006 IEEE

This simple abstraction is capable of some very powerful operations on large data sets. For example, suppose we had a set of web pages, and we would like to build an index of reverse-links; that is, we would like to be able to quickly look up which web pages point to a particular web page. For this task, we write the map function to take as input the contents and address of a web page. When processing page p, for each link to a page q it encounters, the map function emits the pair (q, p). The MapReduce library will then pass pairs of the form (q, P) to the reduce function, where P is the set of all pages that point to page q. The reduce function can then simply output this pair (or perhaps process it somehow for faster lookup).

B. Tornado Codes

A Tornado code [1] is defined by a set of k bipartite graphs $G_1, \ldots, G_k$, where k is some small constant.[1] Each graph G_i has n_i "message" nodes, and m_i "check" nodes. We enforce that the number of message nodes in each graph is equal to the number of check nodes of the previous graph; that is, $n_i = m_{i-1}$ for all $1 < i \leq k$. For a message node i, let $N(i)$ be the neighborhood set of i; similarly let $N(j)$ be the neighborhood set of a check node j.

To encode a binary information word x of length n_1, we place a message bit on each of the n_1 message nodes of G_1. Each check node in G_1 computes a "check bit" by adding together (mod 2) all the message bits in its neighborhood. These $m_1 = n_2$ check bits are passed on the next graph G_2, which uses them to compute its m_2 check bits in the same manner. This continues for all k "layers," and the overall (systematic) code is the concatenation of the original n_1 message bits, as well as the $\sum_i m_i$ check bits computed by the graphs.

The decoding algorithm processes the graphs in the reverse order, first trying to compute the missing message bits of G_k using the (possibly incomplete) m_k check bits of G_k. It then tries to compute the missing message bits of G_{k-1} using the (again, possibly incomplete) $m_{k-1} = n_k$ bits computed by the previous layer, and so on until the first layer G_1 is reached. Thus, the basic decoding task is to compute as many missing message bits as possible of a graph G_i using a set of available check bits of G_i.

This basic decoding task is accomplished via the following simple algorithm, as given in [1]: "Given the value of a check bit and all but one of the message bits on which it depends, set the missing message bit to be the `xor` of the check bit and its known message bits." This algorithm is applied repeatedly until no check bits satisfy this condition.

The key component of Tornado codes is how the graphs G_i are chosen. If they are chosen randomly according to a good degree distribution, then they can approach the capacity of the erasure channel. Specifically, each graph has an associated pair (λ^i, ρ^i) where $\lambda^i, \rho^i \in \mathbb{R}^d$, and d is the maximum degree of the graph G_i. The value λ^i_j represents the probability that a message node in G_i has degree j, and ρ^i_j is the same for check nodes. In [1] they give both theoretical results proving that there exist degree distributions that approach capacity, as well as useful techniques for constructing good degree distributions in practice. In this report, we simply treat the degree distributions as given.

To actually choose the random graph with n_i message nodes and m_i check nodes, we first apply the degree distribution to both sides of the graph, creating the right number of "edge slots" on each node. Then, a random permutation is chosen to match the edge slots on the two sides of the graph.[2]

III. Handling an Enormous Tornado Code

Code Construction and Representation. To make the code truly scalable, we should not necessarily expect to have random access to the representation of the code. So, we need to think about how to represent the code compactly, while preserving the global structure that gives it its error-correcting ability.

In the case of Tornado Codes, we need to be able to represent the bipartite graphs that separate the different layers of the code. Both the encoding and decoding processes will need to perform neighborhood queries in this graph. Specifically, given an index i of a node, we need to have fast access to the indices of the nodes in $N(j)$. Based only on the degree distribution and the index i, we can easily calculate the degree of the given node i, as well as the indices of the "edge slots" associated with that node. Additionally, given the index of an "edge slot" on the opposite side of the graph, we can derive the index of the node to which it is connected. Thus, we need only have quick access to the permutation, as well as the inverse of the permutation, chosen for this graph.

Specifically, we would like a truly random permutation $\pi : \{1, \ldots, N\} \mapsto \{1, \ldots, N\}$ where N is the number of edges in the graph. In practice, we pick a *pseudo-random* permutation given by an access oracle, that can, for any index i, quickly find $\pi(i)$ and $\pi^{-1}(i)$. Ideally, the oracle should be fast and memory efficient. There is tight connection between pseudo-random permutations and private-key block cryptosystems [7]. A block cipher with block size k is essentially a permutation mapping k-bit strings (messages) to k-bit strings (ciphertext). Using a standard block cipher such as DES [8] with block size $k = \lceil \log_2 N \rceil$ should yield a good random-looking permutation.

We assume that our unencoded information x is broken up into equal-sized chunks x_i, and that each one is tagged with its index i. These chunks are consistent with the way the Google File System [9] stores data. Using chunks rather than bits also ensures a minimal overhead in carrying around the index i. For the purposes of encoding and decoding, we can think of each chunk as a logical "bit," since the only operations we will perform on the chunks will be to `xor` two of them

[1] Here we do not have a final block code on the last encoding layer, as is the case with the "theoretical" Tornado codes in [1]. The codes we use are more in line with the "practical" codes defined in [1], Section V.

[2] This may create multiedges, but these can be thrown out without a problem [1].

together. Thus for the remainder of the section, when we refer to message and check bits, we are implicitly referring to larger chunks of data.

Encoding. Encoding a Tornado code requires k executions of a MapReduce, where k is the number of graphs in the code. Typically the first graph G_1 is significantly larger than the others, so the encoding time will typically be dominated by a single MapReduce over the original data file. The total number of map and reduce operations on a single graph G_i is proportional to the number of edges in G_i.

Specifically, for each layer G_i of the Tornado code in order, we run a MapReduce on the n_i message bits of G_i in order to compute the m_i check bits. The map function takes one tagged message bit (i, x_i) as input, and outputs the pair (j, x_i) for each $j \in N(i)$. The reduce function runs on a pair of the form $(j, \{x_i\}_{i \in N(j)})$: it computes the `xor` of all the bits in $N(j)$, and outputs a pair (j, y_j), where $y_j = \bigoplus_{i \in N(j)} x_i$. These pairs are then used as the input message bits to the next layer of the Tornado code.[3]

Decoding. As in conventional decoding of Tornado codes, we decode one layer at a time, from G_k to G_1. For each layer, we attempt to reconstruct as many of the missing message bits as possible using the known check bits.

Whereas encoding required only one MapReduce per layer, decoding requires multiple MapReduces per layer. Each MapReduce pass represents a single decoding iteration, where all check nodes are tested for the condition that their neighborhood is missing only one message bit (and if so, that bit is computed).

Our decoder (detailed below) will maintain a set X of newly-discovered message bits, as well as a set Y of partially-known check bit neighborhoods. We initialize X as the set of known message bits, and Y as the set of known check bits. Each MapReduce will run a map phase on X that essentially mimics the encoding process, and then a reduce phase on both Y and the output of the map phase in order to discover new values of message bits. These newly discovered bits are then set as the new set X, and the process is repeated.

Initialization: Let I be the set of initially known information bits, and J be the set of initially known check bits. Set $X = \{(i, x_i)\}_{i \in I}$, and set $Y = \{(j, N(j), y_j)\}_{j \in J}$.

Repeat until $X = \emptyset$**:**

1) Run the map function on X: For each $(i, x_i) \in X$, output $(j, (i, x_i))$ for all $j \in N(i)$.

2) Run the reduce function on the output of the map function joined with Y. Thus, the reduce function will run on the partially-computed check bits as well as the new message bits computed in the previous iteration.

More formally, fix a particular check bit j. Let V be the set of values (i, x_i) output by the map function with key j, and let $M \subseteq N(j)$ be the set of indices i present in V. The inputs to the reduce function for check bit j are (j, V) from the map function and (j, S, y_j) from Y, where $S \subseteq N(j)$, and $y_j = \bigoplus_{i \in S} x_i$.

Note that the algorithm maintains the invariant $M \subseteq S$. Note also that S and y_j may be missing if they were not present in Y (i.e., the check j was not known initially).

The reduce function has three cases:

- If S and y_j are missing, or if $M = S$: Do nothing.
- If $|S \backslash M| \geq 2$: Set $y'_j = y_j \oplus (\bigoplus_{i \in M} x_i)$, set $S' = S \backslash M$, and output (j, S', y'_j) into Y'.
- If $|S \setminus M| = 1$: Let ℓ be the index of the message bit in $S \setminus M$. Set $x_\ell = y_j \oplus (\bigoplus_{i \in M} x_i)$, and output (ℓ, x_ℓ) into X'.

3) Remove duplicates from X', and store it as decoded data. Set $X = X'$ and $Y = Y'$.

Upon completion of this procedure, the known message bits are contained in the original set X of bits recovered from the channel, as well as the message bits X' that we recovered during the execution of the algorithm.

IV. Conclusion

We have briefly summarized the application of the Google computing infrastructure to the problem of representing, encoding and decoding a large Tornado code over the erasure channel, using many machines. Our experiments are ongoing, and will be given in a subsequent report. It would be interesting to see if other types of error-correcting codes like Turbo codes could also be scaled to large block lengths using these sorts of techniques.

The author acknowledges Martin Pál for helpful discussions.

References

[1] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, “Efficient erasure correcting codes,” *IEEE Trans. on Information Theory*, pp. 569–584, February 2001.

[2] V. Guruswami and P. Indyk, “Near-optimal linear time codes for unique decoding and new list decodable codes over smaller alphabets,” in *Proc. Symposium on the Theory of Computation (STOC)*, 2002.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: turbo-codes,” *Proc. IEEE International Conf. on Comm. (ICC)*, pp. 1064–1070, May 1993.

[4] R. Gallager, “Low-density parity-check codes,” *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[5] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes.* North-Holland, 1981.

[6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. of the Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[7] M. Luby and C. Rackoff, “Pseudo-random permutation generators and cryptographic composition,” in *Proc. Symposium on the Theory of Computation (STOC)*, 1986.

[8] “DES, the digital encryption standard. See: `en.wikipedia.org/wiki/Data_Encryption_Standard`.”

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *Proc. 19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003.

[3] Note that the reduce operation is associative and commutative with respect to the data portion of its output. This property enables a significant savings of time, using the *Combiner* operation of MapReduce [6].