`Orthogonal Matching Pursuit` with a Parallel Approach over `Spark` Pipelines for Lossy Audio Compression

Tito Caco Curimbaba Spadini

Universidade Federal do ABC

Abstract

This work shows three different approaches to implement a parallel method to the Orthogonal Matching Pursuit algorithm (OMP), wich is a compression algorithm that uses Sparse Dictionary Learning concepts. The first attempt is the parallelization of the OMP itself; the second attempt, the parallelization of the Dictionary only; and the third attemp, the parallelization of the signals to be compressed only. The first and the second attempts were both unsuccessful, because of performance or accuracy issues. The third attempt achieved a reduction of 60% on the execution time for the same sparsity level.

Keywords: orthogonal matching pursuit, sparse dictionary learning, parallel, compression, spark

1. Introduction

1.1. Data Representation and Compression

Regardless of whether it is photos, videos, audios, texts or any other type of data, it can be represented by signals. But, unfortunately, the raw information, i.e., the pure signal composed by all original samples, can occupy a huge space on disk. Large files can be a problem in almost any scenario involving data transmission, processing and storage, specially for embedded systems using wireless communication and low cost hardware. So the compressing step is considerably important, since it allows a given raw information, originally big enough to be a problem to be properly transmited, processed and stored after passed by the process of compression. [1]

Depending on what are the applications to this particular information that will be compressed, the original information is, actually, irrelevant, e.g. a MP3 song uses a lossy method of compression that can achieve a very high compression rates and, even being a lossy method, this can be perfectly acceptable to be used to play music on a mobile device while a person listeng to it during a physical activity. [2]

1.2. Sparse Dictionary Learning

A dictionary is a huge matrix composed by many more columns (called “Atoms” in this context) than lines [3]. It’s possible to represent a specific signal ($\boldsymbol{\mathcal{S}}$) using a dictionary ($\boldsymbol{\mathcal{D}}$) and a coefficients vector ($\boldsymbol{\mathcal{C}}$), as follows:

$$\boldsymbol{\mathcal{S}}_{(m\times h)} = \boldsymbol{\mathcal{D}}_{(m\times n)} \times \boldsymbol{\mathcal{C}}_{(n\times h)}, \tag{1}$$

Email address: `tito.caco@ufabc.edu.br` (Tito Caco Curimbaba Spadini) considering that $n \gg m$, i.e. the number of Atoms (columns of the Dictionary matrix) is way bigger than the number of samples per Atom. In this particular case, as this work is directed to an audio application that uses a mono channel, there will have only one column representing that particular channel for each part of the complete audio signal.

2. Orthogonal Matching Pursuit (OMP)

The OMP is a greedy algorithm that tries to represent the given signal (S) with a given dictionary (D) finding the proportion of each Atom that have to be used, considering a stop criterion, like the sparsity level (k), which is the number of different Atoms used to compose the given signal. The coefficients vector uses non-null values exclusively for the positions corresponding to the Atoms that are used to compose the original vector; all other values are equal to zero. [3]

Algorithm 1: Orthogonal Matching Pursuit (OMP)

```
function OMP ($\boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{S}}, K$);
Input  : Dictionary $\boldsymbol{\mathcal{D}}$, signal $\boldsymbol{\mathcal{S}}$ and sparsity level $K$
Output: Support set $\mathbf{I}_k$ and estimate $\hat{\mathbf{x}}_k$
        Initialization:
            1) Iteration counter $k = 1; I_0 = \phi$
            2) $\mathbf{r}_0 = \mathbf{y}; \alpha_0 = \boldsymbol{\mathcal{D}}^T \mathbf{r}_{k-1}$
  repeat
          $i_k = \arg\max(|\alpha_{k-1}|)$
          $\mathbf{I}_k = \mathbf{I}_{k-1} \cup \{i_k\}$;
          $\hat{\mathbf{x}}_{\mathbf{I}_k} = \boldsymbol{\mathcal{D}}^{\dagger}_{\mathbf{I}_k} \mathbf{y}$;
          $\mathbf{r}_k = \mathbf{y} - \boldsymbol{\mathcal{D}}_{\mathbf{I}_k} \hat{\mathbf{x}}_{\mathbf{I}_k}$;
          $\alpha_k = \boldsymbol{\mathcal{D}}^T_{\tilde{\mathbf{I}}_k} \mathbf{r}_k$;
          $k = k + 1$;
  until ($k \geq K$)
```

The Algorithm 1 (from [4]) is a pseudocode adopted as basis to the parallel version developed to this work.

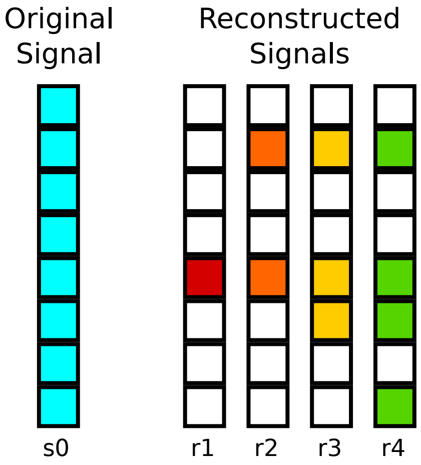

Figure 1: Schematic compressed reconstruction of a signal using OMP algorithm.

The Figure 1 shows a schematic compressed reconstruction of a signal using the OMP algorithm. The original signal ($s0$) is an input of the algorithm and each of the other signals ($r1 \dots r4$) is the result of a step. This is an greedy algorithm, so it will try to get better results each iteration. If the criterion is simply the sparsity leve, the algorithm will stop as soon as it gets the number of non-null components (Atoms) of the Dictionary to compose the representation. And despite the algorithm selects a new component each iteration, keeping all the previously selected components, the respective components coefficient values might not be kept.

3. Parallel Approach

3.1. Big Data and Parallelization

The concept of Big Data can be unterstood as the idea of huge datasets that cannot (or at least should not) be entirely handled by a single computer using a serialized approach. The dataset might be very large, but it doesn't have to processed on the same threads, the same cores, the same processor or even the same machine. Using concepts of parallel computing, it's possible to break the original dataset into multiple chunks that can be processed separately without modifying the data. This can use the available hardware way more efficiently.

3.2. Spark and Parallel Version of OMP

The Apache Spark is an open source distributed computing framework, available in Java, Scala, R and Python. Using pipelines of data, it allows the algorithm to use different processing units simultaneously over Resilient Distributed Datasets (RDDs).

The OMP algorithm cannot be easily parallelized, but there are some specific functions and operations that are significantly more influents on the performance, like matrix transpositions, matrix multiplications and linear regressions. Fortunately, some functions are already implemented for Spark, despite they usually do not accept typical data types as the input.

4. Tests

Environment and Datasets

The test environment followed the hardware specified at the Table 1.

Table 1: Test enviroment.

Part	Description
Processor	Intel Core i7 7700 3.6 GHz (Turbo 4.2 GHz) 8MB Cache
Memory	Kingston HyperX DDR4 16 GB (2 x 8 GB) 2400 MHz
SSD	Kingston UV400 240 GB SATA III
Graphic Card	Nvidia GeForce GTX 1060 6GB

The datasets were composed by almost 100,000 (one hundred thousand) files of acoustic guitar solos, piano solos and acoustic guitar plus piano duets. They follow this configuration: 44100 Hz, 16-bit, 100 ms, WAV. All datasets are private.

4.1. First Attempt

The first attempt was the most naive approach, because I tried to parallelize all core-functions of the OMP. Functions that used a high number of elements with very large matrices was considered here, such as matrix transpose, matrices multiplication and linear regression. Actually, at first, I tried to use the Stochastic Gradient Descent (SGD) to get a parallel (approximated) version of least squares, but this method was even slower and got results that was clearly not properly accurate. So I moved to the idea of ignore the SGD and get back to the use of the simple least squares (from Numpy). Unfortunately, this was terrible in terms of performance.

4.2. Second Attempt

The second attempt was a parallelization of the Dictionary, which is a huge matrix, into some smaller parts, create a RDD and use it at a very simple pipeline, applying directly to a Map function. But here, instead of a bad performance, the problem was about the algorithm itself, because, for each chunk, the algorithm wasn't considering all the atoms from the Dictionary, but only a subdictionary each time, what can make it find the wrong components.

4.3. Third Attempt

Finally, at the third attempt, the signals were parallelized using the RDD over the OMP algorithm applied to the Map function. And this was the best solution at this point in terms of performance, because it achieved $\approx 60\%$ of reduction on the execution time, going from around 2 hours and 2 minutes to around 49 minutes.

5. Conclusion

The best way founded to parallelize this work is to apply the OMP algorithm separately for each subsignal.

References

[1] K. Sayood, Introduction to Data Compression, The Morgan Kaufmann Series in Multimedia Information and Systems, Elsevier Science, 2017.

[2] S. Heath, Multimedia and Communications Technology, Taylor & Francis, 1999.

[3] B. Dumitrescu, P. Irofti, Dictionary Learning Algorithms and Applications, Springer International Publishing, 2018.

[4] J. W. Jhang, Y. H. Huang, A high-snr projection-based atom selection omp processor for compressive sensing, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24 (12) (2016) 3477–3488. doi:10.1109/TVLSI.2016.2554401.