# Large-scale topic modeling

Alexander Frey

sashafrey@gmail.com

## Justification

This project aims is to deliver efficient open-source tools for solving common tasks in topic modeling. This solution must implement few core algorithms (PLSA and LDA), but be efficient on very large collections of documents.

## Goals

* Implement PLSA and LDA
* Use online algorithms, and do not require storing whole doc-word matrix in memory
* Utilize sparsity of doc-word and word-topic matrices
* Scale well for 32 CPU cores and higher, efficiently using shared memory within single process
* Have high convergence rates
* Be portable (written in C/C++, tested with gcc, intel and cl.exe)
* Open-source
* Stable, ready for production usage

## Non-goals

(!) All no-goals might be considered in later versions.

* Distributed cluster solution is a non-goal for v1.0
* CUDA and Intel Xeon Phi support is a non-goal for v1.0
* Support for dynamic dictionary[[1]](#footnote-1) is a non-goal for v1.0.

## Success criteria’s

* The algorithm scales linearly up to 32 CPU cores when process pubmed task (<http://archive.ics.uci.edu/ml/datasets/Bag+of+Words>)
* On small datasets perplexity of our method is on parity with other libraries
* Builds with gcc, intel and cl.exe compiler; runs on Ubuntu, Solaris and Windows.
* No crashes, hangs or memory leaks in stress testing against real-world and model datasets
* Convergence rate is well understood
* Performance model is well understood (Disk, Memory, CPU)

## Design

Execution is divided into two phases: **indexing** and the actual **processing**. The indexing phase produces index files and stores them on disk. Index files expected to have multiple independent **index parts** of aprox. equal size. Each part contains a subset of documents from the whole collection. Processing phase begins with **dataloader** component, which loads index parts (potentially in iterative scans), and store them into in-memory queue. Then **processor** component withdraws index parts from the queue and infers a distribution of the documents into topics. The output is stored in yet another queue. This queue is read by **merger,** which updates word-topic distribution. Before processing new index part the **processor** asks **merger** about the latest view of word-topic matrix.

This design supports multiple concurrent dataloaders, multiple concurrent processors, but only one merger. The system will have the following locks:

1. Lock accesses to the index parts queue (accessed by dataloaders and processors)
2. Lock accesses to doc-topic distribution queue(accessed by processors and merger)
3. Lock for accessing word-topic matrix from processor to merger

It is easy to understand why locks #1 and #2 are cheap: we just assume that index parts are large enough so that the time is mostly spent in reading and processing, not in transferring references from dataloader to processor. Lock #3 will be very check if impose one requirement: once the merger returns a view of word-topic matrix to processor, this view is **read-only** and can be further accessed without locks. Merger will proceed by building a new view (again without locking since it is not in use by any processor), up to the point where new view is different enough from the old view, and hence should be will be replaced with a new one. This will not change the view for all running processors until the finish current index part and request new view from merger. Therefore, lock #3 is only acquired when

a) merger updates the “current view” of world-topic matrix, and

b) processor finishes index part, and requests new view of world-topic matrix.

Therefore all locks #1, #2, #3 are rare and can be implemented as normal boost locks.

This design can be extended to cluster environment by implementing a **networkmerger**, which in addition to merging word-topic matrix within local process should merge it with updates, coming from networkmergers from other nodes.

This design can also utilize CUDA-enabled devices or Index Xeon Phi co-processor by implementing a special processor (without changing the rest of architecture). For CUDA the most promising parallelization is to assign CUDA-threads to topic while inferring a doc-topic distribution.

dataloader might implement caching, so that if the whole collection fits into memory it doesn’t have to reload index parts from disk for the second and further scans.

Data layout:

1. Doc-word matrix: each doc is represented as list of word ids and their counts
2. Word-topic matrix: each word is represented the list of topics and their probabilities
3. Doc-topic matrix: each doc is represented as a sequential vector of topics (no sparsity)

Due to caching in CPU[[2]](#footnote-2) it is important to have topics as a minor dimension in both word-topic and doc-topic matrices.

## Drawbacks of the design

* The doc-topic distribution is calculated each time from scratch, even if document had been processed before and word-topic distribution didn’t change much. Therefore we don’t plan for utilizing sparsity of doc-topic matrix.

1. Dynamic dictionary for online-algorithm means that the dictionary and word-topic matrix is also extended dynamically in run-time. In first version we will assume that the dictionary is known in advance, and the size of word-topic matrix is therefore fixed. [↑](#footnote-ref-1)
2. Read <http://igoro.com/archive/gallery-of-processor-cache-effects/> [↑](#footnote-ref-2)