ssdnm
środowiskowe
studia doktoranckie
z nauk matematycznych

Aleksander Zabłocki

Uniwersytet Warszawski

Distributed computing on large text corpora

Praca semestralna nr 1

(semestr letni 2011/12)

Opiekun pracy: dr Maciej Piasecki (Instytut Informatyki Politechniki Wrocławskiej)

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

The goal of this paper is to study selected cases of using the MapReduce approach (in particular, implemented in Hadoop framework) to allow more effective processing of large amounts of natural language texts. This topic, being relatively new, will certainly gain importance in the coming years as the NLP resources, in particular for Polish language, grow rapidly. Apart from the fact that MapReduce and Hadoop have proved its value by winning a benchmark of sorting one terabyte of data, they are particularly promising from the NLP viewpoint as many linguistic problems seem to suit well into the MapReduce paradigm.

We will focus in particular on SuperMatrix, a system processing large corpora to obtain lexicographic knowledge which can then be used at later stages of text processing. After some evolution, the system recently took a form which allows integrating it with Hadoop, at least from the theoretical point of view. We discuss this issue in detail and present a prototype of Hadooped SuperMatrix which, under certain assumptions, achieves a significant gain in performance.

In Section 1, we acquaint the reader with MapReduce and Hadoop, to an extent which will be needed later. Section 2 presents two successful applications of Hadoop in the NLP field which have done recently. This may be treated either as a loose introduction to later sections or as a set of off-topic remarks.

The main matter is covered in Sections 3 and 4: the former introduces the SuperMatrix system and all its important (from our viewpoint) dependencies, while the latter presents the changed applied to it so far — and a brief discussion of further possible improvements.

1 MapReduce and Hadoop

MapReduce [3] is a general abstract framework for distributed processing of large amounts of data. The input is processed by one or more *jobs*, each of which consists of one *map* and one *reduce* phase, the latter operating on the output of the former. Each phase is performed by a set of *workers* (*mappers* and *reducers*) which can be run at different locations in the cluster; the input for each phase is split and dealt among them.

Both phases are assumed to operate on *key/value pairs*; their input and output are lists of such pairs. It is additionally assumed that the intermediate pairs (i.e. the output of the map phase) are grouped by keys in a way that each group is assigned entirely to one reducer (this is called *shuffling*), and that every reducer receives its input groups sorted by the keys.

The programmer's responsibility is basically providing the code for mappers and reducers, while the rest of data management is left to the implementation of the

framework. However, the user can be given more details to control.

Apache Hadoop is an open-source implementation of the MapReduce framework. It consists of a MapReduce implementation and a dedicated distributed filesystem, HDFS, responsible for managing the data. The MapReduce jobs read from and write to HDFS, and their execution is highly configurable to the programmer. We will now describe this process in more detail.

The filesystem divides each large file into blocks, which can (and usually will) be kept on different nodes. Thus, each block is assigned its own mapper, which will be executed on a node where the block is stored. For both safety and efficiency reasons, the blocks are usually replicated; thus there can be a choice where to start a mapper. As the mappers produce their output, it is partitioned to appropriate reducers and sent to them over the network.

Below we give a list of some important customizable elements of the system.

- The InputFormat partitions the input into InputSplits and selects the cluster nodes where the attached mappers will be executed. The partitioning is a logical operation in which the physical data will usually not participate.
- An InputSplit reads the portion of input assigned to it and converts it to key/value pairs.
- A Combiner (optional) behaves like a reducer but is applied to the output of a single mapper before the data is migrated over the cluster, which can significantly decrease network usage. This could be achieved as well by modifying Mapper, but Combiners are more convenient for the programmer.
- The Partitioner selects the Reducer to receive the group of intermediate pairs with a given key.
- A Comparator governs the sorting order of keys assigned to a concrete Reducer.
- The OutputFormat specifies how the output pairs are saved into files.

Note that the roles of the above elements do partially overlap, with Combiner and Reducer being an apparent example. Notably, the whole functionality of Combiner is also achievable at the level of Mapper, though at the expense of grouping the intermediate pairs manually, which can influence the efficiency.

Being Java-based, Hadoop primarily offers a Java programmer's interface, in which the above elements can be specified as external classes. Hadoop offers two extensions, called Streaming and Pipes, which allow writing own Mappers

and Reducers in other languages; both of them involve some overhead for different variants of the standard Inter-Process Communication (IPC). For the other plug-in classes, integrating with C++ code seems to require invoking manually the Java Native Interface (JNI) or other similar techniques. A reportedly more efficient project, Hadoop C Extension (HCE), is being developed in China; currently it is not publicly available.

2 Applications in NLP

As an introduction to the main matter, we will present two recent applications of the MapReduce paradigm in the NLP field, described in [4] and [5].

2.1 Pairwise document similarity

In [4], the authors aim to compute efficiently a numerical measure of pairwise similarity in a set of documents. Wishing to obtain results as quickly as possible, they choose the following simple formula:

$$sim(d_i, d_j) = \sum_{t \in V} w_{t,d_i} \cdot w_{t,d_j},$$

where d_i denote documents, V is the vocabulary, and w_{t,d_i} denotes the *weight* of a word t in the document d_i, which is taken to be simply the number of occurrences of t in d_i.

Given a set of documents, all the pairwise similarities are computed in two MapReduce jobs: the first creating an inverted index of the documents and the second producing the actual results. We summarize the details below. For legibility, we use the term "k with v" to express that k is a key and v its corresponding value, or a list of values in the case of reducers' input.

Algorithm 1

Map1 Local aggregation: given a word t in document d_i, store it in a dictionary. At the end, emit every t with (d_i, w_{t,d_i}).

Red1 Trivial: given t with a list $[\alpha_1, \ldots, \alpha_k]$, emit the same.
(The list is the index of occurrences of t).

Map2 Given t with $[\alpha_1, \ldots, \alpha_k]$, where $\alpha_s = (d_{i_s}, w_s)$, emit

$$(d_{i_s}, d_{i_t}) \text{ with } w_s \cdot w_t \qquad \text{for all } s \neq t.$$

Red2 Summator: Given (d_i, d_j) with $n_1, \ldots, n_k$, emit (d_i, d_j) with $\sum n_k$.

Note that, at the implementation level, it is not obvious whether to use a Combiner in the **M1** phase. This seems to be not determined in [4]. Note also that, even though the reducer in the **R1** phase does nothing by itself, we cannot abandon this phase because of the shuffle-and-sort step which we rely upon.

The above solution, even though seeming well-suited to the MapReduce principle, is not efficient since the number of pairs emitted after **M2** may be quadratic in terms of the number of documents. As the projects aims in processing the Internet, this number can go into millions, leading to trillions of intermediate pairs. Fortunately, such an explosion occurs only for the most frequent words, which are limited in number by the Zipf's law, and are generally known not to be useful for classifying texts. In [4], it was checked experimentally that removing the 0.1% of most frequent words from consideration speeds up the program by several order of magnitude while not visibly influencing the results, and also makes the problem effectively linear even on million-document web corpora.

2.2 Mavuno — the general setup

A second existing application of Hadoop, namely the Mavuno project [5], is of special interest as it somehow resembles the SuperMatrix system on which we will focus in the next sections. The aim of Mavuno is to detect (and then possibly apply!) synonymies between words using large corpora which can come with various levels of annotation. A common method to do that would be to build a co-occurrence matrix of words and lexico-syntactic *contexts* in which the words could appear, and then guessing the semantic similarity on its basis. However, as the authors wanted to obtain better results, they implemented a more complex method which is basically similar to the Page Rank algorithm.

Recall that in Page Rank the Web, or a part of it, is treated as a directed graph (possibly with edge weights, denoting e.g. the number of links between two

given pages) in which a random walk takes place. More formally, this defines a Markov chain, which is then slightly modified to make it stationary, which means that any initial probability distribution on the state set will eventually converge to a unique invariant limit. This limit distribution is then used for the evaluation of pages.

In Mavuno, a homogeneous graph of the Web is replaced by a bipartite graph of words and contexts, so that a random walk will alternate between them. Also, we are no longer interested in the overall "utility" of vertices, but rather in their similarity to a given word w. Therefore, instead of issuing a random walk from a random vertex, we begin with a probability distribution supported in the vertex corresponding to w. Unlike Page Rank, we do not perturb the Markov chain to become stationary, which allows hoping for meaningful "local" results for different initial vertices. Even if the theory does not promise a success, it is reported in [5] that these expectations are met in practice.

In the language of Markov chains, the transition matrix consists of two null blocks on the diagonal, and two non-zero blocks off the diagonal which describe the occurrences of words in contexts. For an undirected graph, these two blocks would be mutually transpose. However, in Mavuno the graph is directed since the (conditional) "probability" of finding a word w in a context c can differ from the opposite "probability" of finding c around w. We quote here the term "probability" because Mavuno uses various state-of-the-art heuristics as edge weights rather than the actual conditional probabilities.

We are thus left with two separate issues (of which the first is out of our interest):

- determining appropriate edge weights (for accuracy);
- multiplying huge matrices by vectors in MapReduce (for efficiency).

2.3 Matrix multiplication in PageRank and Mavuno

The aforementioned task of multiplying a matrix by a vector is actually performed in the Page Rank algorithm under the assumption that the matrix is sufficiently *sparse*, namely that every its column (assuming that the vector is vertical) can be whole kept in the memory of a worker. However, an abstract formulation of this operation and its assumptions can be hardly found in the literature. We provide them below.

To simplify the procedure, we assume that A and v are stored in the disk in a way which allows the mappers to read pairs

$$(A^1, v_1),\ (A^2, v_2),\ \ldots,\ (A^j, v_j),\ \ldots,$$

where A^i denotes the i-th column of A. (If A and v are kept separately, we can use a custom InputFormat to gain an alternating access to two files). Then the following procedure computes A and $w = A \circ v$ tangled in the same manner:

Algorithm 2

Map Given A^j with v_j, emit

$$j \text{ with } (0,\ A^j) \qquad \text{and} \qquad i \text{ with } (1,\ a_{ij} \cdot v_j) \quad \text{for all } i$$

Red Given i with $\big[(0, A^i),\ (1, \alpha_1),\ \ldots,\ (1, \alpha_j),\ \ldots\big]$, emit

$$A^i \quad \text{with} \quad \alpha_1 + \ldots + \alpha_j + \ldots .$$

Indeed, the last sum is in fact equal to $\sum_j a_{ij} \cdot v_j = w_i$.

In the case of Mavuno, we should ask whether the transition matrices are sufficiently sparse if millions of different words and contexts are to be considered. If all vectors (including A^i) are kept as lists of their non-zero entries, they should all occupy no more than megabytes in current practice, apart from some abnormally frequent exceptions. Such words/contexts may be expected to be removable from the matrix without losing important knowledge, just as in Section 2.1. Note, however, that choosing a (sorted) list presentation of vectors means that vector addition appearing in the **R** phase gets more complicated (and slower).

In [5], the authors follow a somehow different approach. Instead of computing the whole transition matrix A at the beginning, they first look on the given vector v, find its "seriously positive" entries (which are expected to be rare, as they represent "serious candidates" for synonyms), and process the corpus to compute only this part of A which corresponds to those entries.

In exchange for not processing a large matrix A, this method requires scanning the whole corpus separately for each step of the random walk, which is costly. The relation of size between A and the corpus may greatly vary depending on a specific application, in particular, on the variety of possible contexts. These methods can be also synthesized to a memoized solution, in which A^i is computed only when $v_i \neq 0$ and it was not computed during one of the earlier steps. This approach seems to be best in the case where checking words against contexts is costly (which is false in Mavuno, but true e.g. in SuperMatrix).

3 SuperMatrix

SuperMatrix ([2], [1]) is a set of programs and tools which can serve at different stages of deducing semantic relations from a text corpus. In this field, the functionality of SuperMatrix generalizes that of Mavuno. We will now briefly present the ingredients of the system, as well as several other libraries used in SuperMatrix, which will be important in Section 4.

3.1 Poliqarp + Corpus2

Poliqarp [7], originally purposed for searching the IPI PAN Corpus [6] in the XML format, evolved quickly into a system of efficient storing richly annotated corpora in a concise binary format, which was used later e.g. in the National Corpus of Polish. The binary form of a corpus consists of around 40 files of different purpose: the largest one contains the whole corpus data, while the other store several projected images (e.g. only the orthographic forms), the metadata of chunks, or various indexes enabling efficient look-up.

The source code of Poliqarp has been incorporated into the Corpus2 library (a part of Maca project [8]), which provides a dynamic library containing a generic interface for reading and writing corpora in various formats, which invokes the appropriate dedicated readers and writers.

The interface provided by the reader for the Poliqarp format is very limited. After initializing a reader with the path to a corpus directory, the only possible actions are `get_next_token`, `get_next_sequence` and `get_next_chunk`, with a disclaimer (appearing in the source code of Corpus2) that retrieving different kinds of objects from the same reader should be avoided. In particular, since invoking a getter must involve building the output as an object, seeking the corpus is very ineffective (see Fig. 2), regardless of which getter is used.

3.2 WCCL

The Wrocław Corpus Constraint Language (WCCL, [9]) is intended for evaluating compound annotation-dependent expressions over text corpora, primarily in order to extract features for various kinds of machine learning. The expressions are applied positionally (i.e. they always refer to tokens relatively to base position, or a positional variable) and are generally of functional type, although they can have side effects on the internal variables as well as on the annotation in the corpus. (The last features enables performing disambiguation with WCCL).

The real power of the language, however, comes from the set of built-in operations which can be referred to in the user's code. These include various kinds of agreement over a range of tokens, and searching for a token satisfying a given criterion (with saving the position found into a variable). Therefore, WCCL is capable of detecting morpho-syntactic relations across a whole sentence, which was not the case in Mavuno. (As stated in [9], this is particularly important in processing inflectional, free word-order languages, particularly Slavic).

Before the development of WCCL, SuperMatrix based on its predecessor, JOSKIPI. Both tools allow precompiling complex expressions; however, WCCL allows setting variables values in expressions even in their compiled form. This makes a huge different in practice, where a complicated operator template can be parametrized by one string to obtain thousands of contexts, for example: "the argument is a noun described by the adjective *A*". Therefore, unlike in JOSKIPI, the whole space of contexts to be considered can be exhausted by only a few WCCL operators.

3.3 The core functionality of SuperMatrix

The following list presents the basic pipeline of processing a corpus in SuperMatrix. Almost every step is performed by a separate tool; their paths are also listed.

`tools/architect2/basedict`:

1. Build an index counting the occurrences of lemmas in the corpus.
2. Filter the index according to a frequency threshold and a list of accepted parts of speech.

`tools/architect2/builder`:

3. Build an index of so-called *tuples*, counting the word-in-context occurrences.
   (In the current version, the results of `basedict` are not used here).

`tools/architect2/tuplesproc`:

4. Convert an index as above into a SuperMatrix.

`tools/filtrator2/filtrator2`:

5. Filter the rows/columns of a SuperMatrix according to their frequency, entropy a list of accepted lemmas.

`parallel/similarity/parallelSimilarity`:

6. Compute the similarity of selected/all rows in a SuperMatrix, possibly with a user-defined measure of semantic relatedness (MSR).

It is interesting to consider the input for different steps of the process. The corpus is processed in steps 1 and 3; in the other steps, the input consists of a SuperMatrix, a list of lemmas or a list of tuples. The relation of size of these data — and thus the time needed for corresponding steps — depends on the choice of contexts. As showed in Fig. 5, an example WCCL operator applied to the IPI Corpus (of around 250 million segments) produces only 1.8 million tuples after full aggregation. In such situation, the running time of the whole pipeline is dominated by the `builder`.

Another reason to focus on `builder` comes from studying the current methods of distributing computation in SuperMatrix described in [1], which involve steps 3 and 4 (as well as MSR evaluation, which is out of our interest). As the authors admit, a seemingly natural way to handle step 3 would be to split the corpus across a cluster, process the portions separately and gather the results, just along the MapReduce approach. This was however impossible at the time of writing [1] because the contexts expressed as compiled JOSKIPI operators could not be stored all together in the memory of single workers. In the MapReduce language, the solution chosen in [1] is to treat the operators as the input and the corpus as a shared cache (which must be copied to all cluster nodes). Since the development of WCCL, the above problem has been overcomed, which makes it at least considerable to use Hadoop.

Finally, the current implementation of `builder` performs only a partial aggregation of its results. Specifically, the tuples found during the scan are put into an STL `map` object and flushed to the output after reaching a configurable threshold. This is not desired, as it makes the input to `tuplesproc` several times larger than needed (see Fig. 5). In a Hadoop-based solution, a full aggregation would be automatically performed.

4 Hadoop in SuperMatrix

4.1 The general plan

Basically, our aim is to implement the following MapReduce job in Hadoop.

A MapReduce scheme of `builder`

Init	Precompile the WCCL operators.
Map	Given a sentence from the corpus, match it against the operators. Emit every obtained tuple with 1, after local aggregation.
Red	Aggregate: Given t with $[n_1, n_2, \ldots, n_k]$, emit t with $\sum_i n_i$.

Here by "local aggregation" we mean either storing tuples manually in a dictionary and emitting them with the local number of occurrences (which is what the original `builder` does), or letting a Mapper emit tuples with 1 and introducing Combiners, running the same code as the Reducers. We chose the second option for reasons which will be explained later.

Note that both the initialization routine and the map phase make use of the WCCL library. Also, the libraries of Corpus2 and Poliqarp must be invoked somewhere in the map phase to get the sentences from the input corpus. All these tools are implemented in C++ and provided as shared dynamic libraries. Therefore, we must use either one of C++ extensions to Hadoop or JNI, with the first option being both more elegant and portable. As for efficiency, even though JNI is generally considered faster than IPC, using it intensively may slow down a program much more than if it would be rewritten in Java. This is why we will use it in a buffered manner.

4.2 Reading the corpus in parallel

We now come to the main problem, which is caused by the obscurity of Poliqarp format and its supporting library.

In a typical Hadoop job, an appropriate InputFormat splits the input (of a sequential access) in a close correspondence to its physical partition into blocks (to minimize the data flow inside the cluster), but primarily with respect to its fragments (lines / database records / sentences etc.) which the mappers will treat as atomic. In the case of SuperMatrix, we are forced to create a custom InputFormat recognizing sentence boundaries in the Poliqarp binary format.

However, this is not straightforward since the format does not seem to be publicly documented; there is even no evidence that it is sequential. Even scanning the whole corpus chunk by chunk would not help since the information about position in the file(s) seems to be hidden deeply in the implementation. Also, such action can be costly, as we mentioned in Section 3.1.

In our prototype, we made the above naive plan both applicable and efficient, though certainly not elegant, by the following modifications. First, we enriched the Poliqarp and Corpus2 libraries with functions `skip_(token|sequence|chunk)`, which invoke the Poliqarp client to find the next element but do not ask for its object representation. This speeded up a void scan of the corpus (just jumping over the sentences) by more than 50 times. We then used this procedure to quickly forward every mapper to its desired starting position during its initialization. We also perform one additional void scan in the **Init** phase (i.e. in the InputFormat) to determine the size of the corpus, which is needed to assign its equal portions to the mappers. (The corpus size also seems inaccessible through the original API).

As long as our implementation performs seeking the corpus from its beginning in the mappers, it will require a copy of whole corpus at every cluster node. This means that we have currently achieved no progress comparing to [1] in terms of disk usage, although we did in terms of execution time. Note, however, that this problem is possible to overcome if only the corpus is stored in a sequential format as a sequence of sufficiently small independent chunks. In the case of Poliqarp, if possible at all, this would certainly require further expansion of its standard library.

4.3 Other issues

In this subsection, we discuss some minor changes made in SuperMatrix which also impact its efficiency.

The filesystem. The default filesystem in Hadoop, HDFS, is accessible primarily via a Java-based API. Although it can be enriched by a C++ API, or even mounted as a UNIX filesystem using FUSE, both these interfaces are just JNI wrappers of the standard one, which can cause a loss of efficiency. In the current implementation, since the corpus is anyway required to be copied to all nodes, we keep in the regular UNIX filesystem instead of HDFS. As soon as we manage to lift this requirement, it will become profitable to move the data into HDFS, particularly if we assure that the input is read in big portions, which will reduce the number of JNI invocations.

JNI and ANTLR. JNI is used by loading a native shared dynamic library while a Java process executes. Even though Corpus2 and WCCL are both distributes as such libraries, the latter refers to the ANTLR library, provided by default only in a static version (`libantlr.a`). This results in an error of unsatisfied dependency when using JNI. Fortunately, a dynamic version of `libantlr` can be compiled manually by the sources. (By the way, the sources in version 2.7 do not compile on some Ubuntu machines, even after the `configure` script succeeds, unless some basic includes are added to the code).

No Pipes. Since we are forced to connect the Corpus2 library already in the InputFormat and InputSplits, we decided to integrate this with invoking WCCL in order to reduce the usage of JNI. Thus, instead of the corpus data, the input to Mappers consists of the tuples found in it. This implies that the Mapper can be written entirely in Java, and also that it has nothing left to, apart from local aggregation. In such situation, we decide to rely on Hadoop's default aggregation strategy, i.e. we define a trivial identity Mapper and a Combiner which has simply the same code as the Reducer.

JNI buffering. We investigated how buffering the communication through JNI impacts the efficiency. In the current prototype, the textual representations of the tuples are simply appended (with a delimiter) to an STL string at the native

side and then returned to Java after gathering a threshold number of tuples. We observe that both small and large threshold values cause slight overhead (see Fig. 4). The slowdown for large thresholds may come either from the weakness of our string building/splitting routines or from the fact that it leads to an unbalanced data flow inside Hadoop. As for now, it is not clear which factor prevails.

Parallel operators. In the original implementation, parallelism was achieved by starting a separate process for each WCCL operator (using GNU `parallel`). This means that parsing every sentence in the corpus took place several times, which causes certain overhead; see Fig. 2. As in the new implementation each mapper serves all operators at once, we can spend less time on parsing the corpus.

Scalability. In the next several years, we can expect a need, and a possibility, to execute SuperMatrix on a cluster containing significantly more nodes than the number of operators supplied for the program. As soon as this happens, the domination of implementations which split the corpus among the workers (including our program) over the original `builder` will increase. It must be also remembered that it is scalability where Hadoop shows its best. Even though Hadoop-based applications perform rather badly on megabytes of data, they turn out to win benchmarks for terabyte inputs.

4.4 Experiments

We will now present the results of several simple tests of our prototype implementation, which we named `hdsm` (Hadoop-ed SuperMatrix). We analyze their dependence on the configurable parameters and compare with the original `builder` program.

All experiments described here were performed at Wrocław University of Technology, on 2.8 MHz Intel i7 PC with 20 GB RAM, on which Hadoop has been configured in the pseudo-distributed mode. We did not perform tests on multi-cluster nodes, nor did we measure the efficiency of parallelized `builder`; the results of both are quite predictable on small clusters.

Depending on the test, we were using three corpora of significantly different size: a small press corpus (*wordpress_v2*), a medium-sized one of scientific content (*prace*) and the full version of the IPI Corpus (*kipi*). The size statistics of these corpora are shown in Fig. 1.

We first check how much time is spent on bare scanning of the corpus and how much we save with our extension of the Poliqarp library. The improvement is over 50-fold except for the tiny example.

corpus	Mtokens	Msentences	Kchunks	Mbytes
wordpress_v2	0.43	0.022	0.25	12
prace	12.2	0.84	2.53	182
kipi	250	15.8	357	3255

Figure 1: The size statistics of our test corpora.

corpus	size [Mtok]	`builder`	`hdsm`	**gain**
wordpress_v2	0.43	1.4	0.086	**16x**
prace	12.2	49	0.80	**61x**
kipi	250	786	13	**60x**

Figure 2: The time (in seconds) spent on counting the sentences in a corpus.

Then, we compare both implementations with regard to the time of applying one operator on a single multi-cored machine (see Fig. 3). These assumptions somehow favor `hdsm` over `builder` because only `hdsm` can parallelize computation if one operator is given. In general, `hdsm` is expected to beat `builder` as long as

- either the number of available processors exceeds the number of WCCL operators to apply,
- or the operators differ in complexity so that their assignment to the threads of `builder` could not be well-balanced.

corpus	size [Mtok]	`builder`	`hdsm`		**speed gain**
			CPU	real	
wordpress_v2	0.43	5.5	9.9	21	**-74%**
prace	12.2	169	205	112	**34%**
kipi	250	3154	3786	2119	**33%**

Figure 3: The time (in seconds) spent on applying one operator to a corpus on a single multi-core machine. In `hdsm`, two mappers were used, which makes the "CPU time" (computed automatically by Hadoop) around twice the real time.

Such situations seem to be likely to happen in the future practice. On the other hand, `hdsm` performs much worse on very small corpora, as the results show. In other cases, it may be somewhere behind `builder` due to the Hadoop management overhead, though the difference should be much less significant than in the case when `hdsm` wins. Indeed, `hdsm` is still able to parallelize with respect to the operators as well as `builder` is, so it cannot be beaten so easily.

Notably, introducing Hadoop greatly reduces predictability of the execution time, which can be seen in Fig. 4.

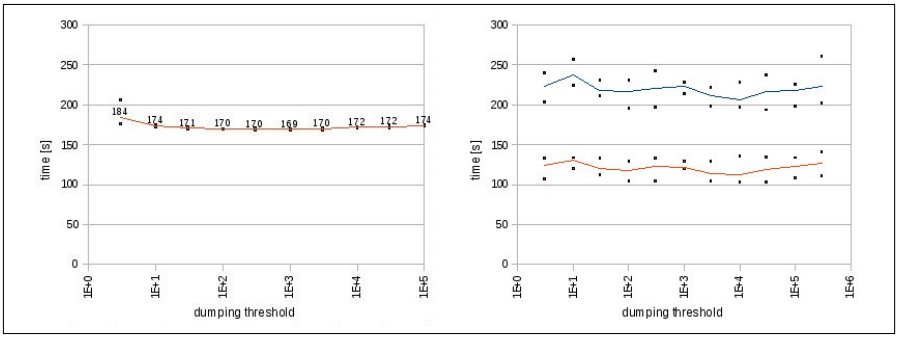

Figure 4: The efficiency of `builder` (left) compared to `hdsm` (right) evaluated with one operator over *prace*, with `hdsm` using two mappers, depending on the local aggregation threshold value. (The graphs are separated because the threshold values in both implementations have somehow different meanings). The real execution times are shown in red, the CPU usage reported by Hadoop is shown in blue. The graphs show averages of five tests, together with minimal and maximal values.

In, Fig. 4 we analyze the impact of the local aggregation threshold on the running time. Even though it is rather insignificant, one can clearly observe that making it larger millions will have a negative impact on efficiency. On the other hand, keeping it even two orders of magnitude below the corpus size may cause an explosion of the output (see Fig. 5). This would in turn influence the efficiency of `tuplesproc` as well as the usability of results in other possible applications.

threshold	`builder`	`hdsm`	**gain**
10K	12		**6.7x**
100K	8	1.8	**4.4x**
1M	4.4		**2.4x**

Figure 5: The time (in seconds) spent on counting the sentences in a corpus.

Conclusion

Trying to fit the functionality of SuperMatrix into the MapReduce paradigm, we obtained significant speed-up (in some situations) by a very simplistic means. The most important improvement was allowing the corpus to be logically split and read in parts by different processes, even though we have not done it from the physical viewpoint and therefore could not lift the requirement of `builder` that the whole corpus be copied everywhere.

Clearly, many possible improvements are still to be implemented and/or investigated. These include:

- physical splitting of Poliqarp binary files, if possible,
- implementing mappers capable of applying multiple operators,
- determining the optimal number of mappers for various cases, (it should certainly grow on multi-node clusters)
- improving the encoding using for JNI communication,
- testing the overhead of C++ wrappers for HDFS,
- porting everything to HCE if it becomes available,

and, last but not least, integrating Hadoop into other parts of SuperMatrix, as soon as the amount of their input data becomes considerably large from the Hadoop's point of view.

Acknowledgements

The author would like to cordially thank Maciej Piasecki, Bartosz Broda and Dominik Piasecki for helpful conversations regarding SuperMatrix. Special thanks go also to Piotr Wieczorek for repeated generous help with Hadoop technical issues.

References

[1] B. Broda, D. Jaworski, M. Piasecki, *Parallel, Massive Processing in SuperMatrix — a General Tool for Distributional Semantic Analysis of Corpus*, Proceedings of the International Multiconference on Computer Science and Information Technology pp. 373–379, 2010.

[2] B. Broda, M. Piasecki, *SuperMatrix: a General Tool for Lexical Semantic Knowledge Acquisition*, Proceedings of the International Multiconference on Computer Science and Information Technology, pp. 345–352, 2007.

[3] J. Dean, S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Sixth Symposium on Operating System Design and Implementation, 2004.

[4] T. Elsayed, J. Lin, D. W. Oard, *Pairwise Document Similarity in Large Collections with MapReduce*, Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers, pp. 265–268, 2008.

[5] D. Metzler, E. Hovy, *Mavuno: A Scalable and Effective Hadoop-Based Paraphrase Acquisition System*, Proceedings of the Third Workshop on Large Scale Data Mining: Theory and Applications, 2011.

[6] A. Przepiórkowski, *Korpus IPI PAN. Wersja wstępna*, IPI PAN, 2004.

[7] A. Przepiórkowski, Z. Krynicki, Ł. Dębowski, M. Woliński, D. Janus, P. Bański, *A Search Tool for Corpora with Positional Tagsets and Ambiguities*, Proceedings of the Fourth International Conference on Language Resources and Evaluation, 2004.

[8] A. Radziszewski, T. Śniatowski, *Maca — a configurable tool to integrate Polish morphological data*, International Workshop on Free/Open-Source Rule-Based Machine Translation, 2011.

[9] A. Radziszewski, T. Śniatowski, A. Wardyński, *WCCL: A Morpho-syntactic Feature Toolkit*, Proceedings of the Balto-Slavonic Natural Language Processing Workshop, 2011.