# High-Performance High-Volume Layered Corpora Annotation

**Tiago Luís and David Martins de Matos**
L$^2$F - INESC-ID
R. Alves Redol 9, Lisboa, Portugal
{tiago.luis,david.matos}@l2f.inesc-id.pt

**Abstract**

NLP systems that deal with large collections of text require significant computational resources, both in terms of space and processing time. Moreover, these systems typically add new layers of linguistic information with references to another layer. The spreading of these layered annotations across different files makes them more difficult to process and access the data. As the amount of input increases, so does the difficulty to process it. One approach is to use distributed parallel computing for solving these larger problems and save time.

We propose a framework that simplifies the integration of independently existing NLP tools to build language-independent NLP systems capable of creating layered annotations. Moreover, it allows the development of scalable NLP systems, that executes NLP tools in parallel, while offering an easy-to-use programming environment and a transparent handling of distributed computing problems. With this framework the execution time was decreased to 40 times less than the original one on a cluster with 80 cores.

## 1 Introduction

Linguistic information can be automatically created by NLP systems. These systems are composed by several NLP tools that are typically executed in a pipeline, where each tool performs a processing step. Therefore, each tool uses the results produced by the previous processing steps and produces new linguistic information that can be later used by other tools. The addition of new layers of linguistic information (layered annotations) by NLP tools makes the processing and access to data difficult due to the spreading of the layered annotations across different files. Moreover, whenever these tools are integrated, several problems related with information flow between them may arise. A given tool may need an annotation previously produced by another tool but some of the information in annotation can be lost in conversions between the different tool data formats, because the expressiveness of each format may be different and not completely convertible into other formats.

Besides tool integration problems, there is also another problem related with the data-intensive nature of NLP and the computation power needed to produce the linguistic information. The wealth of annotations has increased the amount of data to process. Therefore, the processing of this linguistic information is a computation-heavy process and some algorithms continue to take a long time (hours or days) to produce their results. This kind of processing can benefit from distributed parallel computing but it may create other problems, such as fault tolerance to machine failures. Because some NLP algorithms can take long time to produce their results, it is important to automatically recover from these failures, in order not to lose the results of computations already performed. Task scheduling is also a problem due to data-intensive nature of NLP. Data-driven scheduling (based on data location) improves performance because it reduces bandwidth usage.

Our framework aims to simplify the integration of independently developed NLP tools, while providing an easy-to-use programming environment, and transparent handling of distributed computing problems, such as fault tolerance and task scheduling, when executing the NLP tools in parallel. Moreover, NLP systems built on top of the framework are language-independent and produce layered annotations. We also measured the gains that can be achieved with the parallel execution of NLP



*Proceedings of the Third Linguistic Annotation Workshop, ACL-IJCNLP 2009*, pages 99–107,
Suntec, Singapore, 6-7 August 2009. ©2009 ACL and AFNLP

tools and the merging of the layered annotations.

Section 2 discusses related work, Section 3 present the framework's architecture and a detailed description of its components, Section 4 shows the integrated tools, Section 5 explains how the information produced by tools is merged, and Section 6 presents the achieved results. Finally, Section 7 presents concluding remarks.

## 2 Related Work

GATE (Cunningham et al., 2002) is one of the most used framework for building NLP systems. However, it does not provide a controller for parallel execution, it only supports the execution of applications on different machines over data shared on the server (Bontcheva et al., 2004). However, this solution cannot be applied in a large-scale distributed environment because the shared repository becomes a bottleneck in computation due to the accesses from all the machines making computations.

UIMA (Ferrucci and Lally, 2004) is also used to build NLP systems, and this framework supports replication of pipeline components to improve throughput on multi-processor or multi-machine platforms. However, we did not find any published results regarding the parallel execution. The UIMA framework has been successfully leveraged (Egner et al., 2007) with Condor[1], a manager of loosely coupled compute resources, allowing the parallel execution of multiple instances of the NLP system built with UIMA. The Condor scheduler allows to solve problems where there is no communication between tasks and complicates the development of parallel applications when this interaction is needed, like in our case, where it is necessary to merge multiple layers of annotations. Also, the Condor does not move computations closer to their input data, like the MapReduce approach.

The MapReduce paradigm has already been successfully adopted by the Ontea semantic annotator (Laclavík et al., 2008). We think that GATE and UIMA frameworks could also benefit with the MapReduce. For example, the NLTK (Loper and Bird, 2002) adopted this paradigm, and already have implementations of some algorithms like term frequency-inverse document frequency (tf-idf) or expectation-maximization (EM).

There are already tools for merging of layered annotations, like the ANC tool (Ide and Suderman, 2006). However, we did not find any approach to this task in a scalable manner.

[1]http://www.cs.wisc.edu/condor/

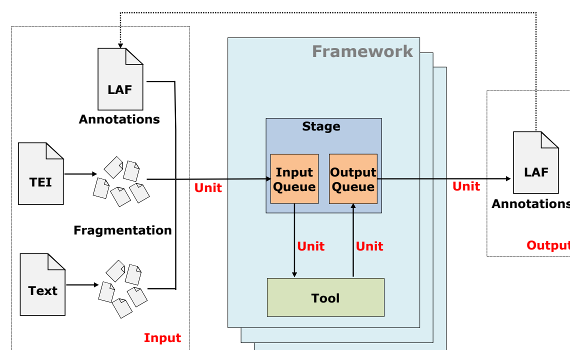


Figure 1: Framework architecture.

Concerning the parallel programming approaches, Message Passing Interface (MPI) (Gropp, 2001) continues to be widely used in parallel programming and therefore there are currently many libraries built based on this programming model. However, this approach provides very low level routines that are difficult to use and make for obscure algorithm implementation, making code reuse and maintenance difficult and time consuming. MPI programming can be difficult because it is necessary to divide the problem among processes with separate address spaces and coordinate these processes with communication routines.

MapReduce (Dean and Ghemawat, 2008) forces the programmer to consider the data parallelism of the computation. Also, this framework automatically schedules and distributes data to tasks. The simple API provided by the system allows programmers to write simple serial programs that are run in a distributed way, while hiding several parallel programming details. Therefore, this framework is accessible to a wide range of developers and allows them to write their applications at a higher level of abstraction than the MPI approach.

## 3 Framework Architecture

Our framework aims to simplify the integration of independently developed NLP tools, while executing the NLP tools in a parallel manner. Our architecture is composed by: Stage, Tool, and Unit (see Figure 1). Stages represent phases in the annotation process of the NLP system. They can be interconnected in order to form an NLP system. The

Tool interacts with the existing NLP tool and can receive as input an annotation previously created or a text file (structured or unstructured). The text files received are divided into sets of independent Units. Units are used to represent the input file fragmentation (each fragment is represented by a Unit). These independent Units are then processed in parallel. Previously created annotations are already divided, since they correspond to an annotation that refers the corresponding input fragment.

Tools are wrapped in Stage components. Stages have two queues: an input and an output queue of Units. Stages are responsible for consuming input queue Units, pass them to the Tool and, after their processing, put the result on output queue. These queues allow multithreaded consumption and production of the Units.

The framework was implemented using the MapReduce (Dean and Ghemawat, 2008) paradigm due to its scalability when dealing with large data volumes. The Hadoop[2] framework (described in the next section) was used as the base for implementation. The next sections describe the representation format used for annotations, the input accepted, and the framework components in more detail.

### 3.1 Hadoop

Hadoop is a MapReduce implementation written in Java. One of the main advantages of using the MapReduce paradigm is task scheduling. When dealing with large datasets in a distributed manner, bandwidth to data becomes a problem. The MapReduce paradigm and the Hadoop Distributed File System (HDFS) allows to reduce bandwidth consumption because tasks are scheduled close to their inputs whenever possible.

Another advantage is fault tolerance and task synchronization handling. These problems, inherent to distributed systems, are transparently solved by the Hadoop framework, facilitating programming of distributed applications.

The MapReduce framework operates exclusively on key/value pairs, i.e., the framework views the input to the job as a set of key/value pairs and produces a set of key/value pairs as the output of the job. These key and value elements can be any user defined data type.

The main tasks of MapReduce are the map and the reduce task. The map task produces a set of intermediate key/value pairs from input key/value pairs. Each map is an individual task that runs on a machine. The reduce phase creates a smaller set of key/value pairs from a set of intermediate values that have the same key. Since different mappers can output the same key, the framework groups reducer input key/value pairs with the same key. This grouping capability is used to merge annotations produced by different tools and that are related with each other, as shown is the Section 5.

[2] `http://hadoop.apache.org/core/`

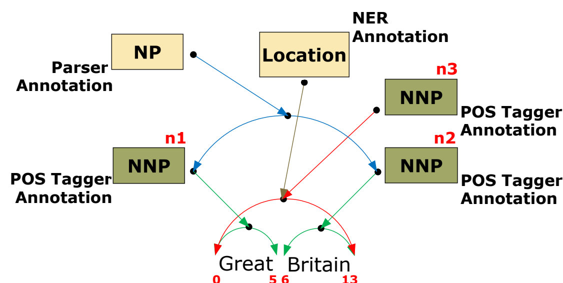


Figure 2: Graph-based model annotation example.

### 3.2 Representation Format

In order to represent linguistic information generated by the tools, we chose the Linguistic Annotation Framework (LAF) (Ide and Romary, 2006) format, that uses a graph model to store annotations.

An annotation can be viewed as a set of linguistic information items that are associated with some data (a part of a text or speech signal, for example), called primary data. Primary data objects are represented by locations in the input. These locations can be the offset of a character comprising a sentence or word, in the case of a text input, or a point at which a given temporal event begins or ends, in the case of a speech signal input. As such, primary data objects have a simple structure. However, it is possible to build more complex data objects, composed by sets of contiguous or noncontiguous locations. Primary data objects are used to build segmentations over data. A segmentation represents a list of ordered segments, where each segment represents a linguistic element. A segment is represented by an edge between virtual nodes located between each character in the primary data (see Figure 2). It is possible to define multiple segmentations over the same primary data, and multiple annotations may refer to the same segmentation.

An annotation is defined as a label and a feature structure. A feature structure is itself a graph in

```
<laf>
    <edgeSet>
        <edge id="e1" from="0" to="5"/>
        <edge id="e2" from="6" to="13"/>
    </edgeSet>
    <nodeSet>
        <node id="n1" edgesTo="e1">
            <fs type="segment">
                <f name="SEGMENT" value="Great"/>
                <f name="POS" value="NNP"/>
            </fs>
        </node>
        <node id="n2" edgesTo="e2">
            <fs type="segment">
                <f name="SEGMENT" value="Britain"/>
                <f name="POS" value="NNP"/>
            </fs>
        </node>
        <node id="n3" edgesTo="e1 e2">
            <fs type="segment">
                <f name="SEGMENT" value="Great Britain"/>
                <f name="POS" value="NNP"/>
            </fs>
        </node>
    </nodeSet>
</laf>
```

Figure 3: Morphosyntactic LAF annotation example.

```
<TEI.2 lang="en" >
    <teiHeader>
        ...
    </ teiHeader >
    <text lang="en" >
        ...
        <p id="p16" >Great</p>
        ...
        <p id="p29" >Britain</p>
        ...
    </text>
</TEI.2>                                TEI file
```

```
<laf addressing="XPointer" >
    <edgeSet>
        <edge id="e1"
            from="xpointer(id("p16")/text()/point[position()=0])"
            to="xpointer(id("p16")/text()/point[position()=5])"/>
        <edge id="e2"
            from="xpointer(id("p29")/text()/point[position()=0])"
            to="xpointer(id("p29")/text()/point[position()=7])"/>
    </edgeSet>
    <nodeSet>
        ...
    </nodeSet>                                      LAF annotation
</laf>
```

Figure 4: TEI file LAF annotation example.

which nodes are labeled with feature/value pairs or other feature structures. Hence, a morphosyntactic annotation is represented by a graph in which nodes are labeled with feature/value pairs. These pairs contain the morphosyntactic information. Figure 3 shows how the two possible segmentations in the POS tagger annotation in Figure 2 can be represented: the segment “Great Britain” has a total of 13 characters; the edges use the character offsets to delimit the segment; the nodes built on top of these edges contain the morphosyntactic information, such as the POS, and the text pointed to by the segment. As shown in the third node (with identifier “n3”), it is possible to have a node referring to multiple edges. A node can also refer to other nodes to add other kinds of linguistic information, such as dependencies between segments or syntactic annotations.

### 3.3 Input

Currently, the Tools integrated in our framework can process three kinds of input files: structured and unstructured text, and previously created annotations. The structured text format currently supported is TEI (Text Encoding Initiative)[3]. Both structured and unstructured text are fragmented into a set of Units. The division is currently paragraph-based, in the case of the unstructured text, and on XML textual elements, in the case of the TEI input. However, it is possible to create Units with other user-defined granularity.

In order to make references to locations in the TEI input, we adopted the XPointer[4] format (see Figure 4). Assuming that each text element in the TEI file has a unique identifier, the XPointer of the start and end tag will refer this identifier and the word character offset.

### 3.4 Unit

When processing large files, with several gigabytes, it is not efficient to process them in serial mode due to memory constraints. Therefore, we divide them into sets of Units that are processed independently.

Each Unit is associated with a portion of the input file and contains the linguistic information generated by a tool in a stage. The Unit has a unique identifier, a set of dependencies (contains information about other Units that the Unit depends on), the identifier of the Stage that produced the unit and the annotation (linguistic information produced by Tool). Besides these elements, it also has a common identifier that is shared across the layered annotations that are related with each other.

[3] http://www.tei-c.org/

[4] http://www.w3.org/TR/xptr/

### 3.5 Stage

Stages represent a phase in the annotation process. Each Stage has two queues: an input and an output queue of Units. This component is responsible for consuming input units, pass them to the Tool and, after their processing, putting them on the output queue. The Units in the output queue can later be used by another Stage (by connecting the output queue to the input queue of the next stage) or written to a file.

An NLP system can be composed of several Stages that are responsible for a specific annotation task. The framework allows the composition of various Tools to form a complete NLP system: each Tool receives the information produced by the Tools in the previous Stages and produces a Unit with the annotation created with references to the previous ones. This information is maintained in memory, along the created tool pipeline, and is only written to disk at the end of the NLP system.

### 3.6 Tool

Tools are responsible for specific linguistic tasks. Currently, these Tools include (without limitation) Tokenizers and Classifiers. Tokenizers receive the input text and produce segmentations (list of segments) that refer to the input, i.e., divide the input sentences into words. Classifiers produce sets of classifications for a given segmentation. These classifications can be, for example, the grammar class of each word. These tools accept two kinds of inputs: an input text or a previously created annotation with a segmentation.

In order to add new tools, it is necessary to extend the previous classes and add the necessary code in order to add the information produced by the existing NLP tool in the LAF format.

Because the framework is written in Java, and the tools could have been developed in a different language, such as C++ or Perl, it was necessary to find a way to interact with other programming languages. Hence, an existing tool can be integrated in various ways. If a tool provides an API, we currently provide an Remote Procedure Call (RPC) mechanism with the Thrift[5] software library. If the API can be used in a C/C++ program, it is also possible to use the existing tool API with Java Native Interface (JNI) (Liang, 1999). The framework also supports tools that can only be executed from the command line.

[5]http://incubator.apache.org/thrift/

## 4 Applications

The tools that have been integrated can be divided into two classes: those capable of producing first level annotations and those capable of producing second level annotations. The first level tools produce morphosyntactic annotation from an input text. Second level tools receive morphosyntactic information as input and produce morphosyntactic annotations. To show the language independence of the framework, we integrated tools from four different languages: Arabic, Japanese, English, and Portuguese. One of the tools capable of analyzing Arabic texts is AraMorph (Buckwalter, 2002), a Java-based Arabic morphological analyzer. For the Japanese language we chose Chasen (Matsumoto et al., 1999), a morphological analyzer capable of processing Japanese texts. The Stanford POS Tagger (Toutanova, 2000; Toutanova et al., 2003) is only being used to process English texts but it can be easily adapted (by changing its input dictionary) to process other languages, like Chinese or German. For processing Portuguese, we chose the Palavroso morphological analyzer (Medeiros, 1995). The morphological analyzers previously described produce first level annotations, i.e., they receive text as input and produce annotations.

Besides these tools, we also integrated types of tools for testing second level annotations: RuDriCo (Paulo, 2001) and JMARv (Ribeiro et al., 2003). RuDriCo is a post-morphological analyzer that rewrites the results of a morphological analyzer. RuDriCo uses declarative transformation rules based on pattern matching. JMARv is a tool that performs morphosyntactic disambiguation (selects a classification from the possible classifications in each segment from the input sequence). The two previous tools were used for processing Portuguese morphosyntactic information, but can be easily adapted to process other languages. For example, JMARv could be used to disambiguate AraMorph classifications and RuDriCo could translate Chasen's Japanese POS information into other formats.

## 5 Merging Layered Annotations

The stand-off annotation provided by the LAF format allows to add new layers of linguistic information by creating a tree whose nodes are references to another layer. This approach offers many advantages, like the possibility to distribute the an-

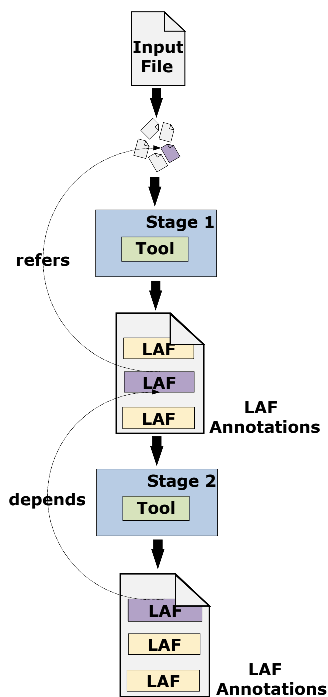


Figure 5: Illustration of the dependencies between annotation layers.

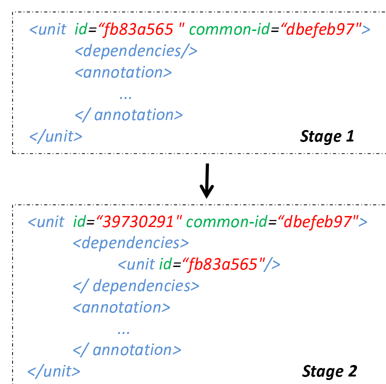


Figure 6: Codification of the layered annotations dependencies.

notations without the source text, and the possibility to annotate discontinuous segments of text. However, these layers, although separate, depend on each other, and their information can be difficult to access, because these layers can be spread across different files. There is a naïve approach to do this merge, that consists in loading all annotations from all files to memory and then resolve their dependencies. However, these dependencies can be dispersed across several large files (see Figure 5. Thus, the machine memory constraints become a problem for this solution.

Therefore, we propose a novel solution to solve the merging problem in a efficient manner using the MapReduce programming paradigm. The grouping capability offered by the Hadoop framework – a Java implementation of the MapReduce paradigm – allows to efficiently merge the annotations produced by the different tools, i.e., the layered annotations. This operation is performed as follows:

**Map** - this phase produces key/value pairs with a key equal to the identifier that is shared by annotations that depend on one another (see Figure 6). Thus, all related annotations are grouped by the framework after this phase.

**Reduce** - before the creation of the new annotation, merge the previously created annotations. This merging process creates a single annotation that contains all the annotations that were combined. This unified annotation is then passed to the Tool. The Tool processes the annotation and produces another one sharing a common identifier. The new annotation is written at the end of this phase.

The serialization of the intermediate key and value elements from a pair in a binary format allows us to reduce bandwidth usage due to the more compact representation of the key and value compared to the LAF (XML-based format representation of the input file).

## 6 Results

The tests were performed on a cluster with 20 machines. Each machine had an Intel Quad-Core Q6600 2.4 GHz processor, 8 GB of DDR2 RAM at 667 MHz and was connected to a gigabit ethernet.

To measure the amount of achieved parallelism, we used the speedup formula shown in Equation 1: $T_s$ is the execution time of the sequential algorithm; $T_p$ is the execution time of the parallel algorithm. Speedup refers to how much a parallel solution is faster than the corresponding sequential solution.

$$S = \frac{T_s}{T_p} \tag{1}$$

The Hadoop framework was installed on all machines and each one was configured to run 4 map and 4 reduce tasks simultaneously. The Hadoop uses HDFS as storage. This file system was configured to split each file into 64 MB chunks. These blocks are replicated across the machines in the

| Data [MB] | Stanford POS Tagger Serial Time [s] |
|---|---|
| 1 | 308 |
| 2 | 606 |
| 5 | 1531 |
| 10 | 3055 |
| 20 | 6021 |
| 50 | 15253 |

Table 1: Serial processing time of the Stanford POS Tagger

cluster in order to tolerate machine failures. We used a HDFS replication factor of three.

To test the system, we used the speedup formula and selected the Stanford POS Tagger. Table 1 shows the serial execution time of the Stanford POS Tagger. This time corresponds to the standalone execution of the tool (without being integrated in the framework) on a single computer, for various sizes of input data (from 1 MB to 50 MB). Input and output were read/written from/to the local disk.

In addition to the previous tool, we also tested JMARv in order to assess the impact of annotation merging at execution time. Unlike the other tools, this tool receives annotations as input.

We must also consider the setup time for Hadoop. When executing the tools on top of Hadoop, it is necessary to store the input data on HDFS. However, these files are, in many cases, rarely updated. Therefore, they are perfect for the write-once read-many nature of HDFS and the copy times of the input data files were not considered (the HDFS write speed was around 22 MB/s).

Section 6.1 shows the speedups achieved with the Stanford POS Tagger, and Section 6.2 the annotation merging results, with the JMARv tool.

### 6.1 Stanford POS Tagger Results

Figure 7 shows the speedup values when considering various values for the number of mappers and reducers, without any compression of the final output, for an input of 50 MB. The large standalone times show that this tool is computationally heavy. With this tool it was possible to achieve a speedup value of approximately 40.

The horizontal progression of the speedup is explained by the heavy computation performed by the tool. Since processing from the Stanford POS Tagger is performed in mappers, the increase in the number of mappers improves speedup values.

The execution time of this tool is around 400

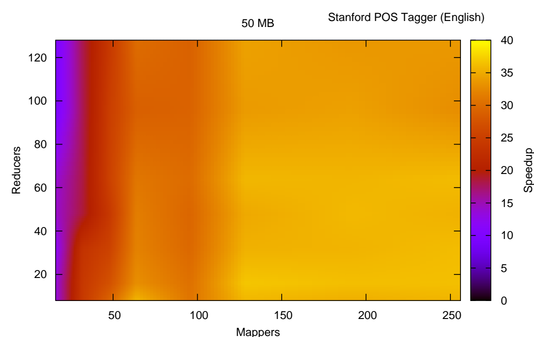


Figure 7: Stanford POS Tagger speedup results

| Input [MB] | Compressed | | Uncompressed | |
|---|---|---|---|---|
| | Output [MB] | Time [s] | Output [MB] | Time [s] |
| 1 | 3 | 56 | 24 | 57 |
| 2 | 6 | 66 | 48 | 67 |
| 5 | 15 | 91 | 119 | 94 |
| 10 | 31 | 140 | 238 | 140 |
| 20 | 62 | 235 | 476 | 226 |
| 50 | 155 | 534 | 1192 | 529 |

Table 2: Stanford POS tagger output compression evaluation with a fixed number of 64 mappers and 64 reducers.

seconds, on the yellow (light gray) portion of Figure 7 and 1700 seconds on the dark blue (black) portions. On the intermediate values the execution time is approximately 1000 seconds.

The top right corner of the graph shows a small speedup decrease. This can be explained by the large number of queued map and reduce tasks.

#### 6.1.1 Compression Evaluation

Table 2 shows how output compression influences execution times values. As shown in Table 2, this tool produces, approximately, an output 24 times larger than the input, without compression, and 3 times larger with compression. However, output compression does not improve execution times due to heavy computation performed by the tool. Hence, processing time dominates output writing time.

### 6.2 Annotation Merging Results

Unlike the previous tool, JMARv does not process text as input. This tool receives an annotation as input that, in this case, was previously created by Palavroso.

In order to test the parallel annotation merging on top of Hadoop, we measured three kinds of times: Palavroso execution time with output cre-

| | Palavroso | Palavroso + JMARv | JMARv |
|---|---|---|---|
| Time [s] | 171 s | 323 s | 179 s |

Table 3: Annotation merging time evaluation with a fixed number of 64 mappers and 64 reducers, for an input of 100 MB of text.

ation time, execution time of JMARv with the previously written Palavroso output and the time of Palavroso and JMARv executed in a pipeline (intermediate data is maintained in memory and the output produced is only written to disk after the execution of the two tools). The results are presented in Table 3. The first column shows the execution time of the Palavroso tool. The second column shows the time of Palavroso and JMARv execution in a pipeline. Finally, the last column shows the execution time of JMARv with the previously created Palavroso output.

In order to execute JMARv after Palavroso, it was necessary to handle about 8 GB of output produced by the previous tool (already stored on disk). However, these results show that running JMARv with this amount of data is practically the same as running both tools in pipeline with the original input (100 MB) and only write their output at the end.

## 7 Conclusions

This framework allowed us to build scalable NLP systems that achieve significant speedups: in the case of computation heavy tools, speedups reached values of approximately 40. In this case, an increase on the number of map tasks improves speedups, because processing time dominates the output writing time.

In addition, the framework supports a wide range of linguistic annotations, thanks to the adoption of LAF. The integration of tools does not consider any aspect related with the parallel execution on top of the Hadoop. Thus, the programmer focuses only on representing the linguistic information produced by the tool for a given input text or previously created annotations. In addition, the programming ease offered by the Hadoop framework allows to focus only on the problem we are solving, i.e., linguistic annotation. All the problems inherent to distributed computing are transparently solved by the platform. The MapReduce sort/grouping capabilities has been used to efficiently merge layered annotations produced by tools integrated in the framework. Regarding future work, on the linguistic part, we plan to integrate tools that produce syntactic annotations (the LAF format already supports these annotations). This linguistic information can be merged with the current tree by simply adding more nodes above the nodes that contain the morphosyntactic annotations. Also, this work did not focus on information normalization. The Data Category Registry (DCR) (Wright, 2004) could be explored in the future, in order to improve interoperability between linguistic resources.

Finally, the creation of NLP systems can be simplified by an XML parametrization. This way it is possible to compose a tool pipeline by simply editing an XML file. An graphical environment for visualization and editing of LAF annotations is also useful.

Our code is available at `http://code.google.com/p/anota/`.

## Acknowledgments

This work was supported by the partnership between Carnegie Mellon University and Portugal's National Science and Technology Foundation (FCT – Fundação para a Ciência e a Tecnologia).

## References

Kalina Bontcheva, Valentin Tablan, Diana Maynard, and Hamish Cunningham. 2004. Evolving gate to meet new challenges in language engineering. *Nat. Lang. Eng.*, 10(3-4):349–373.

Tim Buckwalter. 2002. Buckwalter Arabic Morphological Analyzer Version 1.0. Linguistic Data Consortium, catalog number LDC2002L49, ISBN 1-58563-257-0.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proceedings of the 40th Annual Meeting of the ACL.*

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January.

Michael Thomas Egner, Markus Lorch, and Edd Biddle. 2007. Uima grid: Distributed large-scale text analysis. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 317–326, Washington, DC, USA. IEEE Computer Society.

David Ferrucci and Adam Lally. 2004. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348.

William Gropp. 2001. Learning from the Success of MPI. In Burkhard Monien, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *HiPC*, volume 2228 of *Lecture Notes in Computer Science*, pages 81–94. Springer.

Nancy Ide and Laurent Romary. 2006. Representing linguistic corpora and their annotations. In *Proceedings of the Fifth Language Resources and Evaluation Conference (LREC*.

Nancy Ide and Keith Suderman. 2006. Merging layered annotations. In *Proceedings of Merging and Layering Linguistic Information*, Genoa, Italy.

Michal Laclavík, Martin Šeleng, and Ladislav Hluchý. 2008. Towards large scale semantic annotation built on mapreduce architecture. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 331–338, Berlin, Heidelberg. Springer-Verlag.

Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Edward Loper and Steven Bird. 2002. Nltk: the natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics*, pages 63–70, Morristown, NJ, USA. Association for Computational Linguistics.

Yuji Matsumoto, Akira Kitauchi, Tatsuo Yamashita, Y. Hirano, Hiroshi Matsuda, and Masayuki Asahara, 1999. *Japanese morphological analysis system ChaSen version 2.0 manual 2nd edition*. Nara Institute of Science and Technology, technical report naist-istr99009 edition.

Jos Carlos Medeiros. 1995. Processamento morfolgico e correco ortogrfica do portugułs. Master's thesis, Insituto Superior Tcnico – Universidade Tcnica de Lisboa, Portugal.

Joana Lcio Paulo. 2001. PAsMo - Ps Analisador Morfolgico. Master's thesis, Insituto Superior Tcnico – Universidade Tcnica de Lisboa, Portugal.

Ricardo Ribeiro, Nuno J. Mamede, and Isabel Trancoso. 2003. Using Morphossyntactic Information in TTS Systems: comparing strategies for European Portuguese. In *Computational Processing of the Portuguese Language: 6th International Workshop, PROPOR 2003, Faro, Portugal, June 26-27, 2003. Proceedings*, volume 2721 of *Lecture Notes in Computer Science*. Springer.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL 2003*, pages 252–259.

Kristina Toutanova. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of EMNLP/VLC 2000*, pages 63–70.

S. E. Wright. 2004. A global data category registry for interoperable language resources. In *Proceedings of the Fourth Language Resources and Evaluation Conference – LREC 2004*, pages 123–126. ELRA European Language Resources Association.