Paralelização de Algoritmos de Processamento de Língua Natural em Ambientes Distribuídos

Tiago Manuel da Cruz Luís

Dissertação para obtenção do Grau de Mestre em Engenharia Informática e de Computadores

Júri

Presidente: Doutor Nuno João Neves Mamede
Orientador: Doutor David Manuel Martins de Matos
Vogal: Doutor Luís Manuel Antunes Veiga

Outubro 2008
Agradecimentos

Ao meu pai e à minha mãe a quem dedico este trabalho, pelo carinho e paciência que tiveram comigo durante a sua realização.

Ao David Matos que me acolheu no L²F e que me incentivou a descobrir o mundo da programação paralela. Agradeço também pela orientação na realização deste trabalho.

Ao João Miranda, José Portelo, Aida Cardoso, Vera Mónica, Silvana Abalada, Carlos Rolo, Filipe Martins, Ana Mendes e Paula Vaz pelo bom ambiente criado na melhor sala do L²F e pelo apoio nos momentos mais difíceis, sem esquecer todos os almoços agradáveis e a boa disposição matinal contagiantes da Paula.

Ao Ricardo Ribeiro por todo o apoio e pelas inúmeras perguntas retóricas, das quais se destaca “Já acabaste a tese?”. Ao inigualável professor António Serralheiro pelas visitas relâmpago à nossa sala, sempre cheias de sabedoria humana.

Em suma, gostaria de agradecer a todo o L²F pelo fantástico espírito de equipa e chocolates proporcionados.

Lisboa, November 11, 2008
Tiago Manuel da Cruz Luís
Abstract

Natural language processing (NLP) is a subfield of artificial intelligence and linguistics that studies the problems inherent to the processing of natural language. This area deals with large collections of data that require significant resources, both in terms of space and processing time. Currently, persistent space costs have declined, allowing on the one hand, growth and wealth of description of the data, and on the other hand, increase of the amount of data to process. Despite the fall of persistent storage costs, the processing of these materials is a computation-heavy process and some algorithms continue to take weeks to produce their results.

One of the computation-heavy linguistic tasks is annotation, the process of adding linguistic information to language data or the linguistic annotation itself. Moreover, when these tools are integrated, several problems related with information flow between these tools may arise. For example, a given tool may need an annotation previously produced by another tool but some of this linguistic information can be lost in conversions between the different tool data formats due to differences in expressiveness.

The developed framework simplifies the integration of independently existing NLP tools to form NLP systems without information losses between them. Also, it allows the development of scalable and language-independent NLP systems on top of the Hadoop framework, offering an easy-to-use programming environment and a transparent handling of distributed computing problems, like fault tolerance and task scheduling. With this framework we achieved speedup values around 40 on a cluster with 80 cores.
Resumo

A área de Processamento de Língua natural (PLN) é um subcampo da Inteligência Artificial e Linguística que estuda os problemas inerentes ao processamento e manipulação da linguagem natural. Esta área lida com grandes coleções de dados que exigem recursos significativos, tanto em termos de espaço como de tempo de processamento. Actualmente, o custo do espaço de armazenamento persistente tem vindo a diminuir, permitindo, por um lado, o aumento e enriquecimento da descrição dos dados, e por outro lado, o aumento da quantidade de dados a processar. Apesar da descida dos custos de armazenamento persistente, o processamento destes materiais é um processo computacionalmente pesado e alguns algoritmos continuam a demorar semanas para produzir os seus resultados.

Uma das tarefas computacionalmente pesada é a anotação, o processo de adição de informações linguísticas aos dados linguísticos ou às próprias anotações linguísticas. Quando estas ferramentas são integradas, vários problemas relacionados com o fluxo de informação entre estas ferramentas podem surgir.

A plataforma desenvolvida simplifica a integração de ferramentas de PLN já existentes para formar sistemas de PLN sem perdas informação entre elas. Além disso, permite o desenvolvimento de sistemas de PLN independentes da língua e escaláveis, recorrendo à plataforma Hadoop, oferecendo ao programador um ambiente de programação simplificado. Além disso, permite resolver problemas relacionados com a computação distribuída, como a tolerância a faltas e escalonamento de tarefas, sem qualquer intervenção do programador. Esta plataforma permitiu atingir valores de speedup na ordem dos 40 num cluster com 80 CPUs.
Keywords

Framework
Parallel Programming
Natural Language Processing
Linguistic Annotation
Table of contents

1 Introduction 1
  1.1 Problem ................................................................. 1
  1.2 Objectives ............................................................. 2
  1.3 Structure ............................................................... 2

2 Parallel Programming 5
  2.1 Parallel Programming Models ........................................ 5
      2.1.1 Message Passing ................................................... 5
          2.1.1.1 MPI ............................................................ 6
          2.1.1.2 PVM ............................................................ 6
      2.1.2 Partitioned Global Address Space ................................ 7
          2.1.2.1 Unified Parallel C ............................................ 7
          2.1.2.2 Global Arrays .................................................. 7
      2.1.3 MapReduce .......................................................... 8
          2.1.3.1 Hadoop .......................................................... 9
          2.1.3.2 Dryad ........................................................... 11
      2.1.4 Stream Processing ................................................ 11
      2.1.5 Software Transactional Memory .................................. 12
      2.1.6 Distributed Object-Oriented Model ............................. 12
          2.1.6.1 ProActive ...................................................... 12
      2.1.7 Job Schedulers ..................................................... 13
  2.2 Comparative Summary ................................................ 13
2.2.1 Programming Concerns .............................................. 13
2.2.2 Fault Tolerance ....................................................... 14
2.2.3 Performance ........................................................ 15
2.2.4 Completeness ......................................................... 16
2.2.5 Scheduling ........................................................... 16
2.3 Summary ............................................................... 17

3 Framework Architecture .............................................. 19

3.1 Hadoop ................................................................. 21
  3.1.1 HDFS ............................................................... 22
     3.1.1.1 HDFS Architecture .............................................. 22
     3.1.1.2 Data Replication ................................................ 23
  3.1.2 N-gram Count Example .......................................... 23
3.2 Architecture .......................................................... 26
3.3 Representation Format ............................................... 27
  3.3.1 Linguistic Annotation Framework ............................... 27
     3.3.1.1 Morphosyntactic Annotation ................................. 28
3.4 Processing Unit ...................................................... 30
3.5 Stage ................................................................. 30
3.6 Tool ................................................................. 31
  3.6.1 Tool Execution Mode .............................................. 32
     3.6.1.1 Command Line Execution Mode ......................... 32
     3.6.1.2 Thrift Execution Mode ........................................ 32
     3.6.1.3 JNI Execution Mode ........................................... 33
3.7 Execution ............................................................. 33
  3.7.1 Local Execution .................................................. 34
  3.7.2 Hadoop Execution Mode ........................................ 35
3.7.2.1 Annotation Merging ............................................. 35
3.8 Summary .......................................................... 35

4 Applications ......................................................... 37
4.1 First Level Tools .................................................. 37
  4.1.1 Arabic .......................................................... 37
    4.1.1.1 AraMorph ............................................... 38
  4.1.2 Japanese ....................................................... 38
    4.1.2.1 Chasen .................................................. 40
  4.1.3 English ....................................................... 40
    4.1.3.1 Stanford POS Tagger ................................... 41
  4.1.4 Portuguese ................................................... 41
    4.1.4.1 Palavroso .............................................. 41
4.2 Second Level Tools .............................................. 44
  4.2.1 RuDriCo ....................................................... 44
  4.2.2 JMARv ....................................................... 45
4.3 Summary ........................................................ 48

5 Results ............................................................ 49
5.1 AraMorph Results ............................................... 50
  5.1.1 Compression Evaluation .................................... 52
5.2 Stanford English POS Tagger Results .......................... 52
  5.2.1 Compression Evaluation .................................... 54
5.3 Palavroso Results ............................................... 54
  5.3.1 Compression Evaluation .................................... 56
5.4 JMARv Results ................................................... 57
5.5 Summary ........................................................ 57
6 Conclusions

6.1 Contributions ......................................................... 60

6.2 Future Work .......................................................... 60

6.2.1 Framework Parametrization ..................................... 60

6.2.2 Hadoop Improvements ............................................. 60

6.2.3 Syntactic Analysis .................................................. 61

6.2.4 Graphical Interface ............................................... 61

6.2.5 Tool Integration Improvements ............................... 61

I Appendices ............................................................... 67

A Stanford English POS Tagger ..................................... 69

B Architecture ............................................................ 71
List of Figures

3.1 NLP system using a pipeline architecture ................................................. 19
3.2 NLP system using a shared repository architecture ................................. 20
3.3 N-gram count input file split ................................................................. 24
3.4 N-gram count map phase ........................................................................ 24
3.5 N-gram count reduce phase .................................................................... 25
3.6 N-gram count output ................................................................................. 25
3.7 Framework Architecture .......................................................................... 26
3.8 Framework Class Diagram ....................................................................... 27
3.9 Graph-based model example ..................................................................... 28
3.10 Morphosyntactic annotation example .................................................... 29
3.11 Processing unit diagram .......................................................................... 30
3.12 Stage diagram .......................................................................................... 31
3.13 Command Line execution mode .............................................................. 32
3.14 Thrift execution mode ............................................................................. 33
3.15 JNI execution mode ................................................................................ 33
3.16 Framework Execution Flow ..................................................................... 34
3.17 Annotation dependencies ....................................................................... 34
4.1 Aramorph annotation example ................................................................. 39
4.2 Stanford English POS tagger annotation example .................................... 42
4.3 Palavroso Processing Stages .................................................................... 42
4.4 Palavroso morphological analysis annotation example .......................... 43
List of Tables

5.1 AraMorph serial processing time ........................................ 50
5.2 Aramorph output compression evaluation with a fixed number of 64 mappers and 64 reducers ........................................ 52
5.3 Stanford English POS Tagger serial processing time .............. 52
5.4 Stanford English POS tagger output compression evaluation with a fixed number of 64 mappers and 64 reducers ........................................ 54
5.5 Palavroso serial processing time ........................................ 56
5.6 Palavroso output compression evaluation with a fixed number of 64 mappers and 64 reducers ........................................ 56
5.7 Annotation merging time evaluation with a fixed number of 64 mappers and 64 reducers, for an input of 100 MB of text. .................. 57
The whole industry is betting on parallel computing. They’ve thrown it, but the big problem is catching it. – David Patterson, scientist at the University of California, Berkeley

We live in a time where the computational power needs have grown beyond what the present single-core technology can offer. This technology has several physical and economic constraints such as data transmission speeds through hardware, limits on chip miniaturization and the increased costs of making single processors faster. The evolution of network speeds, distributed systems, and multiprocessor computer architectures shows that parallelism is the future of computing.

Parallel computing allow us to solve larger problems and save time but raises other problems: scheduling of execution of the program across machines, communication and synchronization between them and fault tolerance. Also, debugging and programming is also more difficult than for serial programs.

One of the areas that has large computational needs is natural language processing (NLP). NLP is a subfield of artificial intelligence and linguistics that studies the problems inherent to the processing and manipulation of natural language. This area deals with large collections of data that require significant resources, both in terms of space and processing time. Currently, space costs have declined, allowing on the one hand, growth and wealth of description of the data, and on the other hand, increase of the amount of data to process. Despite the fall of storage costs, the processing of these materials is a computation-heavy process and some algorithms continue to take weeks to produce their results.

1.1 Problem

Annotation, the process of adding linguistic information to language data or the linguistic annotation itself (Ide & Romary, 2004), is one of the computation-heavy linguistic tasks.

NLP systems are composed by several NLP tools that are responsible for creating linguistic information. These systems 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.
When these tools are integrated, several problems related with information flow between them may arise. One of those problems is information being discarded along the chain of components of the NLP system. A given tool may need an annotation previously produced by a 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 data-intensive nature of NLP and the computation power needed to produce the linguistic information. This kind of processing can benefit from parallel computing, but these solutions may create another kind of problems, such as, fault tolerance to machine failures. Because some NLP algorithms can take weeks to produce their results, it is important to automatically recover from this failure, in order to not lose the results of computations already performed.

Task scheduling is also another problem. Data driven scheduling (based on data location) improves performance because it reduces bandwidth usage. This communication overhead reduction is very important in NLP because of their data-intensive nature.

Programming ease is also important in order to focus only on the problem we are solving when developing NLP applications and not on handling distributed computing inherent problems.

## 1.2 Objectives

The main objective of this work is to develop a framework that:

- simplifies the integration of independently developed NLP tools to form NLP systems;
- allows the development of scalable NLP systems that offer an easy-to-use programming environment and a transparent handling of distributed computing problems, such as fault tolerance and task scheduling;
- enables the construction of language-independent NLP systems.

## 1.3 Structure

Chapter 2 presents an overview of parallel programming models applied to clusters (groups of computers), without concerns for domain-related aspects and a comparative summary of presented systems. Chapter 3 describes the developed framework and how it can be used to build NLP systems capable of handling with a large amount of data in a scalable manner. Chapter 4 shows a set of tools that were integrated and chapter 5 presents a discussion of the results obtained.
Finally, chapter 6 makes some remarks about the developed solution and presents some pointers for future work.
Creating software is still something a lot of people can do, but if they had to deal with parallelism, it becomes much more difficult. – Saman Amarasinghe, professor of electrical engineering and computer science at MIT

This chapter presents, in Section 2.1, an overview of parallel programming models applied to clusters (groups of computers), without concerns for domain-related aspects and, in Section 2.2, a comparative summary of presented systems.

2.1 Parallel Programming Models

The programming models are the bridge between the programmer’s natural model of an application and an implementation of that application on available systems. Parallel programming models can be implemented in several ways: as libraries that are invoked from other sequential programs, as language extensions, or completely new execution models.

Parallel computing is a form of computing in which multiple resources are used to solve a computational problem. This type of computing divides large problems into smaller ones that are carried out concurrently (Barney, 2007). One of forms of parallel computing is distributed computing. This type of computing is composed by resources that are connected by a network.

Next sections present an overview of the following programming models on distributed systems: message passing (Section 2.1.1), partitioned global address space (Section 2.1.2), MapReduce (Section 2.1.3), Stream Processing (Section 2.1.4), Software Transactional Memory (Section 2.1.5), Distributed Object-Oriented Model (Section 2.1.6), and Job Schedulers (Section 2.1.7).

2.1.1 Message Passing

Message passing is a form of communication used in distributed parallel programming. In this model, an application runs on a collection of processes, where each one has its own local address space. The processes communicate by sending messages to each other because there is no shared memory space that allows processors to access each other’s memory directly. The message passing operations, such as communication and synchronization between processes, are performed through subroutine calls.
The following sections present an overview of the Message Passing Interface (MPI) and the Parallel Virtual Machine (PVM) message passing specifications.

### 2.1.1.1 MPI

MPI is one of the most successful specifications for message passing. This library specifies a language-independent communication protocol which is widely used in parallel programming in distributed memory. The interface specified by MPI is implemented in various packages, such as Lam (Lam/MPI, n.d.), Mpich (MPICH, n.d.), and OpenMPI (OpenMPI, n.d.).

MPI is widely adopted because it successfully addresses the following properties (Gropp, 2001):

- Portability - a program written for one architecture can be copied to different architectures, compiled and executed without modification;
- Performance - provides an effective way to manage the use of memory and bandwidth;
- Simplicity - organized around a small number of concepts;
- Modularity - designed to support libraries;
- Composability - designed from the beginning to work within a larger collection of software tools;
- Completeness - provides a complete parallel programming model that avoids simplifications that decrease the control the programmer has over the program.

The explicit parallelism and communication provided by MPI is also an advantage because it increases the control the developer has over the application and its performance.

### 2.1.1.2 PVM

PVM is built around the concept of a virtual machine formed by a dynamic collection of (potentially heterogeneous) computational resources managed as a single distributed parallel computer. The virtual machine concept is fundamental to the PVM perspective and provides the basis for heterogeneity, portability, and encapsulation of functions that constitute PVM (Geist et al., 1996).

Unlike MPI, PVM sacrifices some performance in favor of communication flexibility across several architectures and expands the definition of portability to include interoperability. PVM also offers a more flexible task control, allowing the creation and finishing of other tasks during the execution of a program (Geist et al., 1996; Gropp & Lusk, 1997).
2.1.2 Partitioned Global Address Space

Partitioned global address space (PGAS) is a parallel programming model that presents both a private and a shared memory space to the programmer. This model offers the possibility to exploit reference locality because the portions of the shared memory space are associated with particular processes.

While most parallel programs must effectively use many distributed-memory nodes, writing message passing code can be difficult because it is necessary to divide the problem among processes with separate address spaces and coordinate these processes with communication routines (Quammen, 2002). On the other hand, shared-memory paradigms are easier to code and port as serial programs. However, shared-memory has in general poor scalability, because addition of processors in a multiprocessor increases the bus traffic on the system, slowing down memory access time and delaying program execution (Quammen, 2002).

This model is the basis of several language-based implementations such as Unified Parallel C (Yelick et al., 2007), Co-Array Fortan (Co-Array Fortan, n.d.), Titanium (Yelick et al., 2007), and library based implementations such as Global Arrays (Nieplocha et al., 1996). Some of these examples are presented below.

2.1.2.1 Unified Parallel C

Unified Parallel C (UPC) is an extension to the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. UPC adds the following constructs to the C language: an explicitly parallel execution model, a shared address space, synchronization primitives and memory management primitives. In this single shared address space, each variable is associated with a single processor, and can be read and written by any processor. Just like MPI, the amount of parallelism is fixed at program startup time.

Since this approach is based on the extension of a language, it allows the compiler to make code optimizations. However, this has a penalty in programming ease because it forces the programmer to learn a new language.

2.1.2.2 Global Arrays

Global Arrays provides an efficient and portable shared memory programming interface for distributed-memory computers. The parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, through a global index space. This toolkit exposes to the programmer a non-uniform memory access (NUMA) that provides access to a local portion faster than to a
remote one.

This toolkit allows direct access to local portions of shared data, exploiting the data locality information. The goal of Global Arrays is to free programmers from low-level details of communication and to allow them to deal with their problems in the same level they were originally formulated. The library has been designed as a complement the message-passing programming model because programmers can use both shared memory and message passing paradigms in the same program.

2.1.3 MapReduce

MapReduce (Dean, 2006) is a programming model that enables easy and fast development of scalable parallel applications to process large amounts of data on commodity machines. This model expresses distributed computations as sequences of operations distributed over sets of key/value pairs.

A MapReduce computation is composed by two phases, a map phase and a reduce phase, whose inputs are a set of key/value pairs.

- **Map** - for each key/value pair \((K, V)\), the map task invokes a previously defined function that transforms the previous pair into a new intermediate key/value pair \((K', V')\). After the map phase, the framework groups the intermediate key/value pairs with the same keys, sorts them by key and partitions them into a number of fragments equal to the number of reduce tasks.

- **Reduce** - each reduce task consumes a sub-set of the grouped key/value pairs produced in the previous phase \((K', V' \ast)\). For each pair \((K', V' \ast)\), the reduce task invokes a previously defined function that transforms the previous pair into a output key/value pair.

The tasks in each phase are re-assigned to the remaining machines in case of node failure. Having a large number of map and reduce tasks minimizes the overhead of task re-distribution and improves load balancing.

This model has several advantages:

- **Fast development time** - the simple programming API hides parallelization details such as fault tolerance and data distribution;

- **Distributed storage** - data is stored in chunks across machines;

- **Data locality** - moves the computation to the node on which data is stored;

- **Data motion** - it is possible to increase performance by combining intermediary results.
This model is used in various machine learning algorithms, such as, K-Means, Naïve Bayes, Neural Networks, Expectation Maximization and Support Vector Machines (C.T. Chu and S.K. Kim and Y.A. Lin and Y.Y. Yu and G. Bradski and A.Y. Ng and K. Olukotun, 2006). These algorithms are currently being implemented by the recently created project Apache Mahout (Apache Lucene Project, n.d.-). This model is being adopted in several areas, such as machine translation (Dyer, 2007), and different architectures (Kruijf & Sankaralingam, 2007). For example, Google uses this paradigm in their indexing system (Dean & Ghemawat, 2008), that is responsible for producing the data structures used for their web search service, and on collaborative filtering, for generating personalized recommendations for users of Google News (Das et al., 2007).

This model does not directly support processing multiple related heterogeneous datasets. However, it is possible to support this type of processing by adding a merge phase to the MapReduce model: this phase combines data processed by map and reduce phases (Yang et al., 2007). Furthermore, this model does not support communication between the tasks in each phase.

2.1.3.1 Hadoop

Hadoop is a open-source project that implements the MapReduce programming model and has a distributed file system inspired by the Google File System (GFS) (Ghemawat et al., 2003), called Hadoop Distributed File System (HDFS) (Hadoop, n.d.).

HDFS is an highly fault-tolerant and portable file system that runs on commodity hardware and provides a high-throughput on data access. It was designed to reliably store very large files across machines in a large cluster. This file system provides an interface that allows job execution depending on data location. This minimizes network consumption and increases the overall throughput.

Hadoop can also be used with Kosmos Distributed File System (Zvents, n.d.-). This file system has been integrated with Hadoop and, just like HDFS, supports a MapReduce job placement that can schedule jobs on the nodes where the chunks are stored.

Hadoop has been adopted by several organizations, such as Last.fm (LastFM, n.d.), Powerset (Powerset, n.d.) and Yahoo (Yahoo, n.d.) for charts calculation and web log analysis, natural language search, research in advertising systems and web search. For example, on July 2007, Yahoo had more than 5000 nodes running Hadoop, and their biggest cluster had 2000 nodes with 8 cores and 3 TB disk each (Hadoop, n.d.). The following are some examples of applications built on top of Hadoop:

- Nutch - a web search software (Apache Lucene Project, n.d.-);
- Pig - high-level language for expressing data analysis programs (Apache Incubator Project, n.d.);
HBase - distributed storage for managing structured data, that is very effective for storing very large amounts of data in a distributed environment (HBase, n.d.);

Hypertable - high performance distributed data storage system designed to support applications requiring performance, scalability, and reliability (Zvents, n.d.-).

In addition, various frameworks exist to facilitate developing on top of Hadoop, such as Cascading (Cascading, n.d.), a Java library to assist in creating and managing complex MapReduce routines.

The following paragraphs describe two scalable structured data storage systems, HBase and Hypertable, built on top of Hadoop. These systems are inspired by Google Bigtable (Chang et al., 2006) and were designed to store large amounts of data (a typical file in HDFS is gigabytes to terabytes in size). They can also be used with a MapReduce system as an input source and as an output target. There is also a description of Pig, a programming language built on top of Hadoop.

**HBase** is an distributed storage system, inspired by Google Bigtable (Chang et al., 2006), for managing structured data, that is very effective for storing very large amounts of data in a distributed environment. This storage system does not support a full relational model, providing instead a data model that allows the exploration of the locality properties of stored data and dynamically control the data layout and format. Data is stored in a column-oriented form, rather than row-oriented. This storage type allows a better compression ratio, because of data similarity, and space efficiency. HBase can be used as data source and data sink for MapReduce jobs.

**Hypertable** is an high-performance distributed data storage system inspired by Google Bigtable (Chang et al., 2006) designed to support applications requiring performance, scalability, and reliability. Hypertable is designed to manage the storage and processing of information on a large cluster of commodity servers, providing resilience to machine and component failures (Zvents, n.d.-).

This data storage system can run on top of any distributed file system. This initial version of Hypertable supports Hadoop DFS (HDFS) and KosmosFS (Zvents, n.d.-).

Just like HBase, Hypertable is not meant to replace traditional Database Management Systems such as MySQL (MySQL, n.d.) or PostgreSQL (PostgreSQL, n.d.), but rather to be an alternative for storing and processing very large datasets. Traditional RDBMs are transaction-oriented and offer many advanced features for dealing with well-structured data. Hypertable trades off features like joins and advanced querying for scalability and higher throughput. Where row-oriented systems (like MySQL) are in general better for workloads with a lot of write operations, column-oriented systems (like Hypertable) are better for read intensive situations (Zvents, n.d.-).

**Pig** is a high-level programming language targeted to massively parallel processing of large data sets, across clusters, like Google Sawzall (Pike et al., 2005). Both systems are built on top of MapReduce systems and offer a query language to interact with the system.
The Pig query language, Pig Latin, allows the expression of data transformations like group and join, and creation of custom functions, to do specific data transformations. Pig compiles Pig Latin queries into MapReduce jobs and executes them in Hadoop.

Pig can process data in any format and has a data model similar to the relational data model. Unlike in relational models, it is possible to have a table in a field of a tuple. Pig offers several advantages, such as processing of data with different sizes, combination of multiple data sets and code reuse.

However, according to authors, Pig is not adequate for retrieving individual records, or small ranges of records, from a large dataset, and for low-latency data serving requirements (Apache Incubator Project, n.d.).

2.1.3.2 Dryad

Dryad is an general-purpose distributed execution engine for coarse-grain data-parallel applications. Dryad applications combine computational “vertices” with communication “channels” to form dataflow graphs. Dryad runs the application by executing the vertices of a graph on a set of available computers, communicating, as appropriate, through files, TCP pipes, and shared-memory FIFOs (Isard et al., 2007).

The vertices provided by the application are simple sequential programs with no thread creation or locking. Concurrency arises from Dryad scheduling vertices to run simultaneously on multiple computers, or on multiple CPU cores within a computer. The application is able to discover the size and placement of data at run time, and can also modify the graph as the computation progresses to make efficient use of the available resources (Isard et al., 2007).

Dryad is designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. The Dryad execution engine handles all the difficult problems of creating a large distributed, concurrent application: scheduling the use of computers and their CPUs, recovering from communication or computer failures, and transporting data between vertices (Isard et al., 2007).

Dryad application are more powerful than MapReduce’s because it is possible to specify an arbitrary communication DAG rather than a sequence of map, distribute, sort and reduce operations.

2.1.4 Stream Processing

Stream processing is a programming paradigm that expresses computations as a series of operations (kernel functions) applied to each element in the input data (stream).
One of those stream processing systems is StreamIt (Thies et al., 2002), a programming language and a compilation infrastructure, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient and effective mapping to a wide variety of target architectures, including commercial-off-the-shelf uniprocessors, multicore architectures, and clusters of workstations.

2.1.5 Software Transactional Memory

Software Transactional Memory (STM) is a parallel programming model that replaces conventional locks with critical sections expressed as transactions. These transactions control the accesses to the shared memory address space. This approach simplifies the programming difficulty inherent to mutually exclusive locks and barriers synchronization primitives used in parallel applications. Lock-based programming can easily lead to deadlocks, that are difficult to reproduce and debug. The STM is used in single address space parallel machines but can also be used over distributed systems.

One of the distributed STM frameworks is DDSTM (Kotselidis et al., 2008). The DDSTM is a Java-based distributed STM framework that does not rely on any underlying software or hardware shared memory consistency mechanism. The STM is responsible for maintaining the coherence of the distributed memory.

2.1.6 Distributed Object-Oriented Model

This model is composed by distributed and remotely accessible objects that can be accessed through method calls. The ProActive (Baduel et al., 2006) framework is an example of an object-oriented model. There is also Terracotta (Terracotta, n.d.), that enables the deploying of Java applications on multiple JVMs (Java Virtual Machines), yet interact with each other as if they were running on the same JVM. The following section describe ProActive framework.

2.1.6.1 ProActive

ProActive is an Java library that aims to simplify the programming of multithreaded, parallel, and distributed applications for Grids, multi-cores, clusters, and data-centers. It allows concurrent and parallel programming and offers distributed and asynchronous communications, mobility, and a deployment framework.

This system is based on the concept of an active object, which is an entity with its own configurable activity. A distributed or concurrent application built using ProActive is composed of a number of
entities called active objects. Each active object has its own thread of control and method calls sent to active objects are asynchronous.

2.1.7 Job Schedulers

Job schedulers express computations as a set of independent jobs that are bundled together with the instructions necessary to allow them to be processed without intervention. The job schedulers are responsible for managing jobs and provide job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. The jobs submitted to these systems are placed into a queue. Then, the scheduler choose when and where to run the jobs. The system also monitors their progress. There are several job schedulers: Condor (Tannenbaum et al., 2001), PBS (GridWorks, n.d.), Torque (Torque, n.d.), and Sun Grid Engine (Sun Grid Engine, n.d.).

2.2 Comparative Summary

This section presents a comparative summary of several parallel programming features of the models presented in section 2.1.

This comparison aims to evaluate programming ease and code maintenance, fault tolerance support, performance, problem coverage of system (completeness) and task scheduling mechanisms. We studied features like fault tolerance because it is crucial to recover from machine failures since some NLP tasks can take hours or days to produce their results. The problem coverage of system (completeness) is important in order to evaluate the range of problems that can be solved by the programming paradigm. Task scheduling and performance is also crucial in NLP, in order to reduce bandwidth usage. The migration of the computations closer to where the data is located rather than moving the data to where the application is running allows to improve performance in a data-intensive area like NLP.

2.2.1 Programming Concerns

MPI provides low-level routines that are difficult to use and complicates the development of large-scale parallel programs. This also makes algorithm implementation obscure and makes debugging, reuse, and maintenance of the code difficult and time consuming.

PGAS-based approaches simplify code complexity because it offers the possibility of accessing a shared memory space. However, it is necessary to deal with a new problem, memory consistency, because shared memory contexts are affected by the order in which memory operations are made visible to threads. In order to solve this problem, UPC introduces two memory modes, strict and relaxed (Kuchera & Wallace, 2004).
The MapReduce and Dryad frameworks provide a simple API that allows the writing of simple serial programs that are run in a distributed way and that process different sets of data. These frameworks automatically handle several parallel programming issues like fault tolerance and data distribution. This way these frameworks are accessible to a wide class of developers and allow them to write their applications at a larger level of abstraction.

The job schedulers does not provide any API for job communication. Also, the process of division/merging of the input/output data to/from the different jobs must be done by the programmer.

The distributed object-oriented approaches provide an easy-to-use programming environment. The STM shared memory address space simplifies concurrent programming, and the transactions avoid lock-based programming inherent problems, like deadlocks.

The streaming programming style expresses the parallelism inherent in a program by decoupling computation and memory accesses (Gummaraju & Rosenblum, 2005). Therefore, this programming model is more oriented to multicore architectures than distributed systems due to the network latency.

An easy-to-use environment is very important in order to focus only on the problem we are solving when developing NLP applications and not on handling problems inherent to distributed computing.

2.2.2 Fault Tolerance

The MPI specification only specifies fault tolerance for communication channels, which are guaranteed to be reliable, and makes no reference regarding what happens when faults occur in a process. The specification states that the only guaranty after an MPI error is the ability to exit the program (Geist et al., 1996). Nevertheless, there are several MPI implementations, such as MPICH-V (MPICH-V, n.d.), that allow the writing of fault-tolerant programs. MPICH-V provides complete checkpointing and message logging to enable replacement of aborted processes (Gropp & Lusk, 2002).

MapReduce frameworks handle fault tolerance in a automatic manner, totally transparent for the programmer. This simplifies the code complexity and their programming.

Some batch systems does not require a shared file system across machines. With this approach, the batch system transfers the job’s input data files to machines and redirect all the job’s I/O requests back to the submit machine. However, when dealing with large amounts of input and output data, it is not efficient to concentrate the division/merging of the input/output data on a single machine. If a shared file system is available, the batch system can use it to read/write job input/output. However, if the shared file system does not handle fault tolerance, a machine failure can stop all the computations being performed. Also, the computations will depend on shared file system throughput when processing large amounts of data. A small throughput can substantially reduce the speedup value.
Software Transactional Memory research has primarily focused on single address space. STM systems over distributed systems is only starting appearing now, and fault tolerance is an important issue that needs to be addressed (Kotselidis et al., 2008). The object-oriented distributed programming models are much mature and prepared to handle distributed systems problems. For example, ProActive, besides fault-tolerance handling, also load-balances accesses.

NLP computation-heavy processing can benefit from this fault tolerance handling because their algorithms can be very long running. Therefore, if a given machine fails while performing a computation, it is important to automatically recover from this failure.

2.2.3 Performance

The locality principle improves performance because it reduces bandwidth usage. This is particularly important in NLP because usually programs handle large amounts of data.

PGAS implementations, UPC, and Global Arrays, allows the exploration of the locality property. These systems map portions of arrays to each machine and allow them to discover the location of those portions.

Unlike PGAS approaches, MPI implementations do not offer the possibility to explore this property. All data have to be explicitly transmitted across nodes.

Hadoop was designed to deal with large amounts of data and therefore this framework explores the data locality property. One of the HDFS assumptions is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. Hadoop computations are therefore executed near the data they operate on. However, Hadoop has a setup time for storing data under its file system and application start. This last time is caused by application deploying and launch across nodes. Hbase and Hypertable built on top of Hadoop offer scalable solutions to deal with large amounts of structured data that can be integrated with Hadoop MapReduce system.

Job schedulers, unlike MapReduce, does not move computations closer to their input data. In order to achieve this, it is necessary to have a distributed file system that allows queries about the machines that have a given file fragment. Only with this information it is possible to schedule jobs closer to their inputs, if the job scheduler take into account this information.

When dealing with a large amount of data, the distributed object-orient and STM approaches can easily explode to a large number of object/transactions, respectively. Also, in order to move computations closer to their input data, just like job schedulers, it is necessary to change them to take advantage of distributed file system information.
2.2.4 Completeness

MPI has the advantage of being able to implement a large number of parallel algorithms (Gropp, 2001). There are also many libraries based on MPI. One of these libraries is ScaLAPACK (ScaLAPACK, n.d.). This library is important in NLP because of the need for matrix operations e.g. single value decomposition (SVD).

Because of MPI compatibility with Global Arrays, the Global Arrays library can also solve the same kind of problems solved by MPI. Besides that, it also simplifies code complexity related with matrix manipulations.

The MapReduce paradigm does not perfectly fit all kinds of problems. However, if problems are batch oriented (can be run to completion without human interaction), it does not require communication between tasks, and can tolerate a somewhat long startup time, there is a strong possibility that they can fit in this model.

Stream programming perfectly fits on signal processing. However, most programs are not expressed in a streaming-style and have to be recoded, with possibly new algorithms, in order to exploit stream architectures (Gummaraju & Rosenblum, 2005).

The STM and distributed object-orient models can easily solve thread-based problems, by scheduling these execution flow among the machines of the cluster.

Job schedulers allows to solve problems where there is not any communication between jobs (tasks) and difficult the development of parallel applications where this interaction is needed.

2.2.5 Scheduling

MPI- and PGAS-based approaches do not have any mechanism to detect node failures or discovery of new nodes. However, it is possible to integrate MPI implementations with several schedulers such as Condor (Condor Project, n.d.).

Hadoop’s current design assumes that clusters are composed by dedicated servers, and therefore, no scheduling policy exists. The tasks run on each available machine, regardless of their load. However, there is currently a project called Hadoop On Demand (HOD) (Hadoop, n.d.) that aims to integrate this framework with existing schedulers such as Condor (Condor Project, n.d.) and PBS (GridWorks, n.d.) to allow the allocation and provision of nodes dynamically. The HOD project addresses the provisioning and managing of MapReduce instances on cluster resources to allow a fair and efficient use of cluster among different users.

Job schedulers choose the machine where to run the jobs based on a policy, that can be parame-
terized. Also, the user can provide memory and CPU restrictions in their submissions. These systems share the computational resources among many users and programs.

2.3 Summary

MPI continues to be widely used and therefore there are currently many libraries built on top of this programming model. However, this approach provides very low level routines that are difficult to use, obscure algorithm implementation, makes 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 to coordinate these processes using communication routines.

PGAS-based approaches simplify code complexity and does not have the poor scalability of shared-memory systems due to bus traffic congestion that slows down memory access and delays programs execution. The PGAS programming model offers the advantages of the shared memory programming paradigm, but works across distributed memory hardware. However, the latency of the network that connects processors is an issue and therefore it is necessary to find efficient communication schemes. PGAS also introduces a new problem: memory consistency. This problem is caused by the fact that shared-memory context is affected by the order in which memory operations are made visible to threads.

The job schedulers does not provide any API for job communication and leaves the process of division/merging of the input/output data to/from the different jobs to the programmer. Also, these systems does not move computations closer to their input data.

Stream processing model is more oriented to multicore architectures than distributed systems. STM over distributed systems is only starting appearing now, and distributed system inherent problems need to be addressed, in order to increase the acceptance of this approach.

The distributed object-oriented approaches can explode to a large number of objects when dealing with large amounts of data. Just like STM, in order to move computations closer to their input data, it is necessary to change these systems to take advantage of distributed file system information.

The MapReduce and Dryad frameworks force the programmer to consider the data parallelism of the computation. Also, these frameworks automatically schedules and distributes data to tasks. This is important in an area like NLP because it deals with large amounts of data. The simple API provided by the systems allows programmers to write simple serial programs that are run in a distributed way while hiding several parallel programming details. Therefore, these frameworks are accessible to a wide range of developers and allow them to write their applications at a higher level of abstraction than in the MPI and PGAS approaches.
Hence, the Hadoop framework is suitable for handling NLP problems in a scalable manner due to its data-driven task scheduling, fault tolerance automatic handling and programming ease. The other approaches do not handle fault tolerance and scheduling transparently, therefore increasing the difficulty of application development.
Programmers have no choice: if they want fast programs, they’re going to have to write parallel programs – Dave Patterson, director of the UPCRC and computer-science professor at Berkeley

In order to develop scalable NLP systems we develop our framework on top of the Hadoop framework due to their efficient approach for task scheduling (based on data location), automatic handling of fault tolerance and programming ease.

NLP systems are composed by several NLP tools that are responsible for a specific linguistic task. One of those linguistic tasks is annotation, the process of adding linguistic information to language data or the linguistic annotation itself (Ide & Romary, 2004).

There are several kinds of annotations: morphosyntactic annotation (annotation of the grammatical class of each word in the text), syntactic annotation (annotation of the structured sequence of morphosyntactically annotated words) and name entity recognition (subtask of information extraction that seeks to locate and classify atomic elements in text into predefined categories such as the names of persons, organizations, locations or expressions of times), are some examples of possible annotations that can be produced by an NLP tool.

NLP systems are typically executed in a pipeline manner 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, as shown in Figure 3.1.

Whenever these tools are integrated, several problems related with information flow between them may arise, that can cause information losses (Graça, 2006):

- the information discarded along the system may be required further ahead by other tools

![Figure 3.1: NLP system using a pipeline architecture](image-url)
• conversions are necessary between the different data formats because it is common that each tool
define its own data model to represent linguistic information. If the expressiveness of each format
is different, some of the formats may not be completely convertible into other formats

• when viewing tools output as a layer of information over a primary data source and considering
that layers are normally related to each other, it is necessary to maintain relations between those
layers. These relations avoid the repetition of common data and allow the navigation over the data
produced by the different layers

ShRep (Graça, 2006) proposes a solution to these problems. Their solution consists in a client-server
architecture where the server is a shared data store. They propose a shared repository where tools
add new layers of information without changing the existing ones (see Figure 3.2). All the linguistic
information is available in the server. This way, tools only have to select the required information,
avoiding the loading, parsing, and saving of extra data. Since the server is itself a shared data store
information loss is avoided because all information is kept in the repository and it is never removed.
Moreover, the solution avoids the information merging problem, since each tool only uses the layers it
requires as input adding the new produced layer at the end.

However this solution cannot be applied in a distributed environment because the shared reposito-
ry becomes a bottleneck in computation due to the accesses from all the machines making computa-
tions.

We propose a different approach: instead of a client-server architecture, we map all linguistic infor-
mation produced by tools to an abstract model. This abstract model uses a graph model that provides
flexible representation that can be used to represent linguistic annotations. Whenever a tool adds new
linguistic information, a new graph based model is created. Thus, tools still continue to add new lay-
ers of information without changing the existing ones, due to the references from the new graph to the
previous ones, reducing the amount of information created.

However, these dependencies between annotation create a problem that does not exist in ShRep. Due to the spreading of these relative annotations (an annotation can depend of previous created annotations) across various files, it is necessary to merge all these relative annotations into a single one that contains all the linguistic information produced by the tools so far. When dealing with a large amount of annotations, that can have several gigabytes, is not efficient to merge these annotations in a single core due to memory and processing constraints.

Thus we propose a scalable solution to solve this problem, using MapReduce paradigm. Besides the scalable merging, our solution also allows to create scalable NLP systems that are capable of dealing with large amounts of data.

The following sections describes the Hadoop framework (see Section 3.1), the framework architecture (see Section 3.2), and the involved entities.

3.1 Hadoop

Hadoop (Hadoop, n.d.) is a MapReduce (Dean & Ghemawat, 2008) implementation written in Java programming language. Despite the existence of a streaming execution approach for executing the Hadoop applications, that can be used by scripting languages like Ruby or Bash, we have chosen to develop this framework in the Java language and therefore directly use Hadoop’s Java API (Application Programming Interface), in order to have more control over the framework.

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 Hadoop Distributed File System (HDFS) (Hadoop, n.d.) allow us to reduce bandwidth consumption because tasks are scheduled to 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, that is, 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 (class). The only requirement is that these elements must be serializable and deserializable to an output and from an input stream, respectively.

The following paragraphs describe the map and reduce tasks, the partitioner, the combiner, and how input and output can be controlled.
Map - this phase produces a set of intermediate key/value pairs from input key/value pairs. Each map is an individual task that runs on a machine.

Reduce - the reduce phase creates a smaller set of key/value pairs from a set of intermediate values that have the same key. Each reducer has 3 phases: shuffle, sort, and reduce:

- Shuffle - fetches the relevant key/value pairs produced by the map tasks.
- Sort - since different mappers can output the same key, the framework must group reducer input keys. These stage is responsible for the grouping. The shuffle and sort phases occur simultaneously and while map outputs are being fetched they are merged.
- Reduce - receives the previously grouped pairs and produces the final pair(s). The result is typically written to the HDFS filesystem but it is possible to choose a different destination, such as a database management system (DBMS) table update.

Partitioner - controls the partitioning of the keys of the intermediate map outputs, i.e., the assignment of intermediate key/value pairs to the reduce tasks). The partition is based on the key (or a subset of the key). By default the partitioning is done by object hash code. The maximum number of partitions is equal to the number of reduce tasks.

Combiner - performs a local reduce to the map output key/value pairs. This component is very important because this local reducer allows the reduction of network usage due to key/value grouping.

Input Format - controls the input file splitting and converts each one of these splits into a set of key/value pairs.

Output Format - controls the destination to the final key/value pairs.

Section 3.1.1 describes Hadoop Distributed File System (Hadoop, n.d.). In order to illustrate how MapReduce paradigm works, we present an example of an application built on top of the Hadoop framework on Section 3.1.2.

3.1.1 HDFS

The HDFS is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets (Hadoop, n.d.).

3.1.1.1 HDFS Architecture

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a
number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system’s clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode (Hadoop, n.d.).

3.1.1.2 Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time (Hadoop, n.d.).

3.1.2 N-gram Count Example

An N-gram is a sub-sequence of n items from a given sequence. N-gram models predicts \( x_i \) based on \( x_{i-1}, x_{i-2}, \ldots, x_{i-n} \) using \( P(x_i|x_{i-1}, x_{i-2}, \ldots, x_{i-n}) \). N-gram models are widely used in statistical natural language processing.

In this example, we will show how the different components of Hadoop works and can be used to count the occurrences of N-grams with size one (unigrams) and two (bigrams).

Input - before the computations, it is necessary to create the initial key/value pairs of the map phase. For that, we divide the input file in a set of splits that will be assigned to mappers. Note that this file division is done using their size information and not the content. The reading of the whole file to determine this division results in a waste of bandwidth. Then each split is assigned to one mapper. This scheduling, when possible, is done to the machine where the data of the split lies. This data driven scheduling can be improved by increasing the file replication factor. The mapper is then responsible for creating key/value pairs for their computations. In this example, we choose key/value pairs formed by the offset of each line in the file and the line itself (see Figure 3.3).

This division can create splits that do not perfectly match the key/value pairs wanted. For example, the first split in the Figure 3.3 ends at the middle of one line. Therefore our implementation assigns the responsibility of the processing of the line that has been divided to the mapper that
first started their processing. In this example, the first mapper will process the key/value pair for line 3 and the second mapper will start their processing on line 4.

Map - in this case, each mapper is responsible for counting the occurrences of each N-gram. In this example our mapper counts only unigram and bigram occurrences. Therefore, for each key/value pair that represent one line of the input file, they will produce a key/value pair with a unigram or bigram as a key and 1 as value, as shown in Figure 3.4. The count 1 that this unigram or bigram has occurred once. The key/value pairs produced by mappers are stored on the local disk of the machine where each mapper is running. These key/value pairs are already sorted by their key.

Reduce - after the map phase, the framework will group the key/value pairs (produced by all map tasks) with same key. As shown in figure 3.5, each unigram or bigram group will be formed by the unigram or bigram itself and a list of 9 elements with the number one, because our example has 9 identical phrases.

Output - after the reduce phase, it is necessary to choose the destination for the final key/value pairs produced by this phase (Figure 3.6). Each key/value pair is written in an individual file associated with each reducer.
Figure 3.5: N-gram count reduce phase

Figure 3.6: N-gram count output
3.2 Architecture

This section presents the description of the architecture of our framework. Figure 3.7 shows the framework’s architecture and Figure 3.8 shows a simplified view (with the most important components) of the framework class diagram. The most important component of the framework is the Tool. There are currently two kinds of tools: Tokenizer and Classifier. In order to integrate an NLP tool, it is necessary to extend one of the two previous abstract classes and add the necessary code to interact with the actual tool and produce the linguistic information. Tools can receive as input an annotation previously created by another tool or an input text fragment. These two kinds of input are divided into a set of independent units. The Processing Unit is used to represent the input file fragmentation (each fragment is represented by a Processing Unit). These independent units are then processed in parallel.

Since the framework is developed in Java, in order to interact with already existing NLP tools that were developed in a different language, we created the Tool Execution Mode entity.

Tools are wrapped in Stage components. Stages represent a phase in the annotation process of the NLP system and have two queues: an input and an output queue of processing 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 on both queues.
3.3 Representation Format

This section presents a description of the representation format that has been chosen to store annotations. This format must be very flexible to allow the representation of a wide range of linguistic information varieties, like morphosyntactic and syntactic annotations.

There are several formats available. One of those formats is MAF (Morphosyntactic Annotation Framework) (ISO/TC37/SC4, n.d.). MAF is currently being developed by ISO TC 37/SC4. A MAF-annotated document is formed by the original document and a set of annotations. These annotations are represented by word forms covering a set of tokens of the original document. The word form contains the morphosyntactic content and this information is expressed by Feature Structure format (ISO/TC37/SC4, n.d.). One of problems in morphosyntactic annotation is ambiguity due to the different grammatical classes that a word can have. MAF represents ambiguity by organizing tokens and word forms into one or more flow that are materialized by directed acyclic graphs. However, the format only allows the representation of morphosyntactic annotations and it is difficult to add more complex annotations such as syntactic annotations.

LAF (Linguistic Annotation Framework) (ISO/TC37/SC4, n.d.) is another format available. This format was chosen due to its flexibility. This format also uses a graph model to store annotations. The annotations can be viewed as a connected set of more elementary sub-annotations.

3.3.1 Linguistic Annotation Framework

LAF (Linguistic Annotation Framework) is also currently being developed by ISO TC 37/SC4.

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 input. These locations can be an 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 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 a set 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 3.9). 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 which nodes are labeled with feature value pairs or other feature structures.

The following subsection describes how we added morphosyntactic annotation using the LAF representation format.

### 3.3.1.1 Morphosyntactic Annotation

A morphosyntactic annotation is represented by a graph in which nodes are labeled with feature value pairs. These pairs contain the morphosyntactic information. Figure 3.10 shows how the two possible segmentations in the first level annotation, in Figure 3.9, 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 (part-of-speech), and the text pointed 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.
Figure 3.10: Morphosyntactic annotation example
3.4 Processing Unit

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

Each processing unit is associated with a portion of the input file and contain the linguistic information generated by a tool in a stage. The division is currently paragraph-based but it possible to create units with other granularity.

Each processing unit has the following associated information:

- identifier - unique identifier used to reference the processing unit;
- dependencies - contains information about the processing units that the unit depends on;
- stage - identifier of the stage that produced the unit;
- common identifier - identifier shared by all processing units that depend on each other;
- annotation - linguistic information produced by the tool.

Figure 3.11 shows an example of two annotations produced by two tools. The first annotation depends of the input file and has a identifier that is referred to by the annotation produced by the next tool. These dependencies allow the second tool to produce a new annotation with references to the previous one. In this way we save disk space when storing the information.

3.5 Stage

Stages represent a phase in the annotation process. Each stage has two queues: an input and an output queue of processing units. This component is responsible for consuming input queue units, pass them to
the tool (described in Section 3.6) and, after their processing, put them on output queue (see Figure 3.12). The units in output queue can latter be used by another stage or written to a file.

An NLP system can be composed of several stages that are responsible for a specific annotation task. Our framework allows the composition of various tools to form an NLP system. Each tool receives all information produced by the tools in the previous stages and produces a processing 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. These queues allow multithreaded consumption and production of the units in both queues.

**3.6 Tool**

Tools are responsible for specific linguistic tasks. Currently, it is possible to create NLP systems with two kinds of tools: Tokenizers and Classifiers.

A Tokenizer receives the input text and produces a segmentation (list of segments) that refer to the input.

A Classifier produces a set of classifications for a given segmentation. This tool accepts two kinds of inputs: an input text or a previously created annotation with a segmentation, as shown in Figure 3.11.

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.

Because the framework is written in Java, and the tools could have been developed in a different language, like C++ or Perl, it was necessary to find a way to interact with other programming languages. Hence, we created a tool execution mode entity that contains all the logic that allows the interaction with the tools that have already been developed in other languages.

The following subsection presents a description of the created tool execution modes.
3.6.1 Tool Execution Mode

A previously created tool can be integrated in various forms. If a tool provides an API, we currently provide a RPC (Remote Procedure Call) mechanism with Thrift (Thrift, n.d.) software library. If the API can be used in a C/C++ program, it is also possible to use the existing tool API with JNI (Java Native Interface) execution mode (Liang, 1999). However, if the tools can only be executed from the command line, we offer an execution mode capable of executing this type of tools. Besides these execution modes, we also offer another one capable of executing Java based tools. The next three subsections describe command line, Thrift, and JNI tool execution modes.

3.6.1.1 Command Line Execution Mode

This tool execution mode can be used when the tool does not provide an API and can only be executed from the command line. Hence, we spawn a different process to execute the tool. The process input and output streams are then used to send input and receive output respectively (see Figure 3.13).

3.6.1.2 Thrift Execution Mode

Thrift is a software library and set of code-generation tools developed at Facebook (Facebook, n.d.) to expedite development and implementation of efficient and scalable backend services. The library generates all the necessary code to build RPC clients and servers based on a single language-neutral file that describes data types and services interfaces.

The RPC client is in the framework and the RPC server interacts with the tool’s API. The Thrift code generation engine allows building services that work efficiently between Java and C++, C#, Perl, Python, PHP, Erlang, and Ruby. Figure 3.14 illustrates these interactions: the RPC server is spawned in a process, loads the tool on startup, only once, and reuses the reference to the tool for RPC client invocations, in order to reduce interaction time.
3.6.1.3 JNI Execution Mode

The Java Native Interface (JNI) (Liang, 1999) enables the integration of code written in Java with code written in C and C++. It allows programmers to take full advantage of the Java platform without having to abandon their investment in previously created code (see Figure 3.15).

This approach, unlike Thrift, does not have the network communication overhead, although the communication between RPC client and server is locally executed. However, there is a performance overhead in the JVM (Java Virtual Machine) due to security checks before invoking JNI code. Besides this, Java applications that depend on the JNI can no longer readily run on heterogeneous environments.

Like in the Thrift execution mode, we also store a reference to the object that represents the tool. In this way, we avoid the initial setup time of the tools in each invocation, reducing the invocation time.

3.7 Execution

This section describes the two NLP systems execution modes that are currently supported by the framework: local execution and Hadoop execution mode. The first mode corresponds to the execution of the tools locally, on a single machine, and the second to parallel execution, on a machine cluster, on top of the Hadoop framework, using the MapReduce paradigm. Figure 3.16 shows the execution flow of the
previously described entities. In both execution modes, it is necessary to combine the different annotations produced by tools. Section 3.7.1 shows the typical and inefficient solution for this problem and Section 3.7.2 shows our scalable approach to the annotation merging problem.

### 3.7.1 Local Execution

The local execution mode executes the tools locally and merges the annotations using a naive approach. The Figure 3.17 illustrates this process. The first annotation produced contains a reference to the input file fragment. If the first annotation is passed to a second tool, it will create a second one, that depends of the previous one. Hence, it is necessary to merge these related annotations if we want to invoke a new tool. To perform this operation, in the local execution, we load all the annotations to memory and then resolve their dependencies. However, these dependencies can be dispersed across several large files. Thus, the machine memory constraints become a problem in this kind of processing.

![Figure 3.17: Annotation dependencies](image-url)
3.7.2 Hadoop Execution Mode

The Hadoop execution mode allows the execution of tools in parallel, on a machine cluster, using the Hadoop framework, and perform annotation merging in a scalable way.

Depending on the input, annotation creation can be made in two ways. When processing an input data text, the annotation computation, by the tools, is performed in the map tasks. However, if the input is composed by previously created annotations, the annotation creation is done on reduce tasks, as described before. This difference is explained by the need of merging the previously created annotations (that can be spread across several files), in order to pass this information to tool.

3.7.2.1 Annotation Merging

The merge and sort capabilities offered by the Hadoop framework allow us to efficiently merge annotations produced by the different tools.

The following paragraphs describe how map and reduce tasks perform this operation.

Map - this phase produces key/value pairs with a key equal to the identifier that is shared by annotations that depend on one another. This way, all related annotations are grouped by the framework after this phase.

Reduce - before the creation of the new annotation, merges the previous related annotations. This merging creates a single annotation that contains all the annotations that were combined. This information is then passed to the tool. The tool processes this annotation and produces another one that shares a common identifier. The new annotation is then written at the end of this phase.

The serialization of the intermediate key and value elements from a pair allows us to reduce bandwidth usage due to the more compact representation of the key and value compared to the XML representation of the input file. This representation also allows a more efficient parsing than an XML representation.

The intermediate key and value elements can also be compressed by the framework using the Gzip (GNU Zip) or the LZO (Lempel-Ziv-Oberhumer) compression algorithms, in order to reduce even more the data transferred between tasks.

3.8 Summary

This chapter presented a description of the components of the Java-based framework that has been developed as a solution for the distributed annotation problem. This framework allows us to create scalable NLP systems, and facilitates the integration of tools that can be written in another language.
The representation format chosen for storing annotations is very flexible due to the graph-based model representation of linguistic annotation.

This chapter also presented the framework’s implementation, on top of the Hadoop framework, and how MapReduce can be used to merge annotations produced by the different tools.
We are dedicating all of our future product development to multicore designs. This is a sea change in computing. – Paul Otellini, president of Intel

This chapter presents a description of the tools that have been integrated in the framework. All these tools produce morphosyntactic information from a previous annotation or from an input text. In order to integrate them in the framework, they all have extended the Classifier abstract class and use the existing Tool Execution Modes in order to obtain the linguistic information produced by the existing tools.

The tools that have been integrated can be divided into two classes: first level tools and second level tools. The first ones produce morphosyntactic annotations from an input text. The second ones receive morphosyntactic information as input and produce morphosyntactic annotations. The following sections presents the tools that have integrated in both classes.

### 4.1 First Level Tools

To show the language independence of the framework, we integrated tools from four different languages: Arabic, Japanese, English, and Portuguese. Section 4.1.1 describes Arabic morphological analysis, Section 4.1.2 describes Japanese processing, Section 4.1.3 describes English analysis, and section 4.1.4 Portuguese processing.

All these tools receive text as input and produce morphosyntactic annotations.

#### 4.1.1 Arabic

Arabic language is spoken in more than 20 countries and it is the native language of over 200 million people. Arabic morphemes (smallest linguistic unit that has semantic meaning) are defined by three consonants, to which various affixes (prefixes, suffixes and infixes) can be attached to create a word.

One of the tools capable of analyzing Arabic texts is AraMorph. The following subsection describes this tool and how we represent the morphosyntactic information.
4.1.1 AraMorph

AraMorph is a Java port of the Arabic morphological analyzer developed in Perl by Tim Buckwalter (Buckwalter Arabic Morphological Analyzer, n.d.).

Arabic words are defined as one or more contiguous Arabic characters. Non-Arabic strings are split on white space and left unanalyzed. Due to the Arabic orthography, AraMorph determines the best tokenization by doing a statistical analysis of the entire input text.

After the tokenization step, AraMorph segments the words into prefix, stem (part of a word that is common to all its inflected variants), and suffix strings. AraMorph then performs the following operations after tokenization and word segmentation, previously described:

- dictionary lookup and compatibility check – for each segmentation, verifies if the prefix exists in the prefixes hash table, the stem exists in the stems hash table and the suffix exists in the suffixes hash table. If all three components are found in their respective hash tables, determines if their morphological categories are compatible;
- second lookup (orthographic variants) – when a word returns no analysis, this tool checks the orthography of the input string and creates a list of alternate orthographic forms.

Tool Integration

Figure 4.1 shows an example of an annotation produced by this tool for an input example. We are currently representing only a subset of the information produced by this tool. We chose to only represent stem POS and lemma information, but the prefix and the suffix information can be easily added in the future.

The edges in Figure 4.1 delimit words in input text by their offsets. These offsets are given by the AraMorph tool. The nodes that refer to these edges are then used to represent morphosyntactic information.

As shown in node 1 of the Figure 4.1 (that refers to edge 1), this tool produces ambiguous morphosyntactic information. The multiple feature structure elements (represented by the “fs” XML element) allows the ambiguity to be represented.

4.1.2 Japanese

Japanese language is spoken by more than 130 million people. Japanese is an agglutinative language, i.e., it forms words by putting together basic elements, called morphemes, that retain their original forms
Figure 4.1: Aramorph annotation example
and meanings. A morpheme is a distinctive linguistic unit of relatively stable meaning that cannot be divided into smaller meaningful parts.

Chasen (Y. Matsumoto, A. Kitauchi, T. Yamashita, Y. Hirano, H. Matsuda, K. Takaoka, M. Asahara, n.d.) is a morphological analyzer capable of processing Japanese texts. The following section describes this tool and how it was integrated in the framework.

4.1.2.1 Chasen

ChaSen is a morphological analysis system that segments Japanese sentences into morphemes and tags those morphemes with their parts of speech and pronunciations.

For a given Japanese input sentence, Chasen consults its morpheme dictionaries and records all the possible morphemes that are any sub-strings of the input string. Chasen then calculates the morpheme costs and connectivity costs, sums these value and outputs results with minimum cost (Y. Matsumoto, A. Kitauchi, T. Yamashita, Y. Hirano, H. Matsuda, K. Takaoka, M. Asahara, n.d.).

The connectivity of two morphemes / parts of speech is defined in the form of their bigrams. In order to tune its costs, part of speech bigram Markov model is employed, and the probability parameters of Maximum Likelihood Estimate (MLE) model is transformed into its connectivity costs. The costs of morphemes are also obtained from the MLE model (Y. Matsumoto, A. Kitauchi, T. Yamashita, Y. Hirano, H. Matsuda, K. Takaoka, M. Asahara, n.d.).

Tool Integration

In order to integrate this tool in the framework, we extended the Classifier abstract class and used the Command Line tool execution mode. In this way, we launch the Chasen tool on an external process and send Japanese input text into its input stream. The process output stream is used to collect the resulting information.

This tool provides a programming API that is not currently exploited. Also, we are only representing the segmentation produced.

4.1.3 English

English language is spoken in more than 50 countries and it is the native language of over 400 million people. The total number of speakers is more than 1.8 billion.

There are several tools capable of analyzing English texts but we choose to integrate only one tool. The Stanford English POS Tagger (described in subsection 4.1.3.1) is one of the most used POS taggers.
Currently, this POS tagger is only being used to process English texts but it can be easily adapted (by changing their input dictionary) to process other languages, like Chinese or German.

4.1.3.1 Stanford POS Tagger

The Stanford log-linear English POS tagger is based on Maximum Entropy Markov models (Toutanova, 2000; Toutanova et al., 2003). Using the maximum entropy method, the tagger learns a conditional probability model from tagged text. The model assigns a probability for every tag in the set of possible tags given a word and its context, which is usually defined as the sequence of several words and tags preceding the word. This model can then be used for estimating the probability of a tag sequence for a given a sentence.

Tool Integration

The integration of this POS tagger was facilitated by its Java API. In order to integrate this tool on the framework, it was only necessary to extend the Classifier abstract class and use the Native execution mode. In this way, it was not necessary to launch any external process in order to invoke the tool’s API.

Figure 4.2 shows an annotation example for the input phrase “quick brown fox”. This tool segments the input text and only produces the segment’s POS information. Thus, the “fs” XML element contains only the segment text and its POS.

4.1.4 Portuguese

The Portuguese language is spoken in more than 9 countries and it is the native language of over 240 million people. The following section describe Palavroso tool.

4.1.4.1 Palavroso

Palavroso (Medeiros, 1995) is a morphological analyzer that produces a set of classifications for each segment in input sequence. The processing done by this tool is composed by three steps:

- identification of the words that belong to the input sequence;
- application of morphological rules to each word, producing a set of their hypothetical classifications;
- confirmation of classifications with the dictionary.
Figure 4.2: Stanford English POS tagger annotation example

Figure 4.3: Palavroso Processing Stages
Figure 4.4: Palavroso morphological analysis annotation example
This tool can produce a response with all produced classifications or only with the classifications that were confirmed by the dictionary, as shown in Figure 4.3.

After the identification of the word “almôco”, the application of morphological rules produces three possible classifications (“almôco”, “almoçar” and “almoçer”). In the last phase of the processing, Palavroso verifies which of these classifications are confirmed by the dictionary. In this case, only the classifications for “almôco” and “almoçar” are confirmed.

**Tool Integration**

Palavroso provides a C programming API. Therefore, we used this API in Thrift execution mode. We also obtain the information produced by this tool using Command Line execution mode. The JNI could also be used but is not currently implemented.

Figure 4.4 shows an annotation example produced by this tool. The segmentation information produced is represented by a set of edges that refers to locations in the input. The morphosyntactic information is added on top of these elements.

### 4.2 Second Level Tools

The following tools are currently being used for processing Portuguese morphosyntactic information but they can be easily adapted to process all the other languages, by changing their dictionaries and rules. For example, JMARv could be used to disambiguate AraMorph classifications and RuDriCo could translate Chasen Japanese POS information into English.

Section 4.2.1 presents a description of RuDriCo tool and Section 4.2.2 describes JMARv.

These tools receive morphosyntactic information as input and produces morphosyntactic annotations.

#### 4.2.1 RuDriCo

RuDriCo is a post-morphological analyzer, based on PAsMo (Pardal, 2001), that rewrites the results of a morphological analyzer. RuDriCo uses declarative transformation rules based on pattern matching. Thus, it allows to:

- change the segmentation made by the morphological analyzer (for example, changing ‘nas’ into ‘em’ and ‘as’; or gluing dates like ‘2’, ‘de’ and ‘Janeiro’ into ‘2 de Janeiro’; always constructing features for the new segments from the existing features);
• change the information associated to the words tagged by the morphological analyzer (for example, solving some ambiguity by removing inaccurate tags given to a segment by the morphological analyzer);

• change the output format of the morphological analyzer in order to make it suitable to be used by the next tool (by outputting the segments in a different format according to the given rules).

RuDriCo is developed in C++ and provides a programming API. Hence, we used the library provided to invoke this tool.

However, the integration of this tool raised a performance problem due to its API. On JNI and Thrift execution modes we created a single object that represents this tool and it is used to invoke RuDriCo. However, each invocation receives an input stream for the input XML document and returns an output stream for the output XML document, produced by the tool. The performance could be improved by replacing these streams by an object representation of their content, in order to eliminate the XML conversions.

Tool integration

RuDriCo is a post-morphological analyzer, that besides changing the morphological information, also creates a new segmentation for input.

The new segmentation created is represented by a new “edgeSet” XML element. The morphosyntactic information is then added in the nodes that points to the edges, in “fs” XML element.

Currently we are replicating some edges to same data. In the future we pretend to merge the edges that point to same segment. This way, RuDriCo will only add inexistent segments to the previous segmentations, thus reducing the amount of information created.

Figure 4.5 shows an output example for the annotation produced by Palavroso, represented in Figure 4.4.

4.2.2 JMARv

JMARv, unlike all the other integrated tools, was created during this work. JMARv is a java port of MARv (Ribeiro, 2003) tool and can be integrated in NLP systems that need to perform morphosyntactic disambiguation (selects a classification from the possible classifications in each segment from the input sequence). JMARv’s input is composed by a list of segments, where each segment is annotated with all possible classifications. JMARv probabilistic disambiguation module reduces the possible classifications and produces the input sequence completely disambiguated.
Figure 4.5: RuDriCo annotation example
Figure 4.6: JMARv annotation example
The following subsection presents a description of the probabilistic disambiguation module.

**Probabilistic disambiguation module**

This module is based on Markov models (Ribeiro, 2003) and uses the Viterbi algorithm (Ribeiro, 2003) to find the most probable sequence of classifications \( t_1, t_2, \ldots, t_n \) for the input sequence \( w_1, w_2, \ldots, w_n \).

\[
\arg \max_{t_1, \ldots, t_n} \prod_{i=1}^{n} P(t_i|t_{i-2}, t_{i-1}) P(w_i|t_i)
\]  

(4.1)

The probabilities \( P(t_i|t_{i-2}, t_{i-1}) \) encode relative contextual information of the labels and the probabilities \( P(w_i|t_i) \) encode lexical information.

The Viterbi algorithm is then used to find the most probable classification sequence for that input (Ribeiro, 2003).

**Tool integration**

The integration of this tool was simple, since it was implementation in Java. The current implementation selects the last existing annotation and disambiguates their possible classifications. As shown in Figure 4.6, the disambiguated classification refers to the segments produced by the previous tool (see Figure 4.5).

This dependency from the previous annotation is included on the meta information associated to the annotations. This meta information is included in the Processing Unit element.

**4.3 Summary**

This chapter presented an overview of the integrated tools, for four different languages. We showed that our framework supports the representation of a wide range of linguistic information varieties. The Tool Execution Modes also supported the integration of these tools.

When integrating these tools, it is not necessary to consider any aspect related with the parallel execution on top of the Hadoop. The programmer focuses only on representing the linguistic information produced by tool for a given input text or a previously created annotation.
Parallelism is coming not just to high-end systems, but across a very broad slice of computer science, and we expect to permeate every corner of computer science going forward. – Andrew Chien, director of Intel Research

This chapter shows the results obtained for a subset of the integrated tools. The tests were performed on a cluster with 20 machines. Each machine has an Intel Quad-Core Q6600 processor, 8 GB of DDR2 RAM at 667 MHz and is connected to a gigabit ethernet. The Hadoop framework is installed in these machines and each one has been configured to run 8 map and 8 reduce tasks simultaneously.

The Hadoop framework uses HDFS as storage. This file system has been configured to split each file into 64 MB chunks. These blocks are replicated across machines in the cluster in order to tolerate machine failures. The HDFS replication factor is three.

During these tests, the users continued to perform their daily tasks that sometimes occupied all the machine cores. Also, some machines crashed during the tests. Although our system automatically recovered from these failures, these times were not considered because they introduce noise in the results.

To measure the amount of parallelism achieved, we used the speedup formula. Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. Speedup is defined by the following formula:

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

(5.1) where:

- $T_s$ is the execution time of the sequential algorithm
- $T_p$ is the execution time of the parallel algorithm processors

To test the developed system we selected a subset of the integrated tools. The Stanford English POS tagger is one of the selected tools due to its slow speed, compared with the other tools. AraMorph and Palavroso were also selected due to their very fast morphological analysis and the amount of information produced. All the annotation process in these tools, when running the parallel tests on top of the Hadoop, were performed on map tasks, because these tools process text as input.
Besides the previous tools, we also tested JMARv in order to test the impact of annotation merging in execution time. Unlike the other tools, this tool receives annotations as input. Therefore, the annotation creation (by tools) is performed on reduce task.

Section 5.1 describes the results for AraMorph tool, Section 5.2 shows the speedups achieved in the Stanford English POS Tagger, Section 5.3 the Palavroso results, and Section 5.4 the JMARv results.

## 5.1 AraMorph Results

To compute the speedups achieved, we measured the serial execution times (Table 5.1). These times correspond to the standalone execution of the AraMorph tool (without being integrated on the framework) on a single computer. Figure 5.1 shows the speedup values for each combination of the number of mappers and reducers, without any compression of the final output. Speedups greater than 1 only start appearing for inputs greater than 5 MB, due to the overhead of launching parallel tasks. The maximum speedup is achieved when the number of mappers is close to the number of cores available (in total, there are 80 cores available). When the number of mappers is greater than the number of cores in the cluster, the speedup decreases (as shown in the speedup graph for the 100 MB input text, in Figure 5.1). This can be explained by the large number of map tasks in the queue. For example, for the input text of 100 MB, each map task is processing an average of 1 MB if the number of map tasks is equal to 100. Therefore, the overhead of launching this amount of tasks overwhelms the processing time. Thus, it is better to process an average of 2 MB, for the same input, with 50 mappers.

The optimum number of reducers is also around the number of cores available. This way we get a higher write throughput of the generated output for the HDFS.

The execution times of this tool is around 140 seconds, on the yellow portion (light gray on black-white scale) of graphs on Figure 5.1 and 270 seconds on the dark blue portions (black on black-white scale). On the red portion (dark gray on black-white scale) the execution times are around 200 seconds.

The following subsection discusses the impact of output compression, using Gzip, on execution

<table>
<thead>
<tr>
<th>Input [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>124</td>
</tr>
<tr>
<td>10</td>
<td>245</td>
</tr>
<tr>
<td>20</td>
<td>484</td>
</tr>
<tr>
<td>50</td>
<td>1188</td>
</tr>
<tr>
<td>100</td>
<td>2400</td>
</tr>
</tbody>
</table>

Table 5.1: AraMorph serial processing time
Figure 5.1: AraMorph POS Tagger speedup results
5.1.1 Compression Evaluation

Table 5.2 shows a comparison of the output generated and execution times with and without compression. All these results were obtained with a fixed number of 64 map tasks and 64 reduce tasks. The compression of the final output with Gzip reduces the execution for inputs greater than 20 MB. With compression, the output generated is, on average, 3 times bigger than the input, and not 30 times as in the case where no compression is used. This way it is possible to continue to increase the input data in a scalable manner.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>34</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>67</td>
<td>50</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>167</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>334</td>
<td>58</td>
<td>57</td>
</tr>
<tr>
<td>20</td>
<td>60</td>
<td>662</td>
<td>64</td>
<td>68</td>
</tr>
<tr>
<td>50</td>
<td>149</td>
<td>1655</td>
<td>90</td>
<td>87</td>
</tr>
<tr>
<td>100</td>
<td>295</td>
<td>3310</td>
<td>132</td>
<td>145</td>
</tr>
</tbody>
</table>

Table 5.2: Aramorph output compression evaluation with a fixed number of 64 mappers and 64 reducers

<table>
<thead>
<tr>
<th>Data [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>308</td>
</tr>
<tr>
<td>2</td>
<td>606</td>
</tr>
<tr>
<td>5</td>
<td>1531</td>
</tr>
<tr>
<td>10</td>
<td>3055</td>
</tr>
<tr>
<td>20</td>
<td>6021</td>
</tr>
<tr>
<td>50</td>
<td>15253</td>
</tr>
</tbody>
</table>

Table 5.3: Stanford English POS Tagger serial processing time times.

5.2 Stanford English POS Tagger Results

Table 5.3 shows Stanford English POS tagger serial results. These times correspond, once again, to the standalone execution of the Stanford English POS Tagger tool (without being integrated in the framework) on a single computer. Figure 5.2 shows the speedup values with mappers and reducers variation, without any compression of the final output. The large standalone times shows that this tools is computation heavy. Thus, we achieved a large speedup value (around 40), with speedups greater than 1 started appearing sooner, even for small inputs.
Figure 5.2: Stanford English POS Tagger speedup results
The last graph in Figure 5.2 shows the evolution of speedup values for an input text of 50 MB. The horizontal progression of the speedup is explained by the heavy computation performed by this tool. Since processing is performed in mappers, the increase in the number of improves speedup values.

The execution time of this tool is around 400 seconds, on the yellow portion (light gray on black-white scale) of graphs on Figure 5.2 and 1700 seconds on the dark blue portions (black on black-white scale). On the red portion (dark gray on black-white scale) the execution times are around 1000 seconds.

On the top right corner of the graphs appears a small speedup decrease. This can be explained by the large number of queued map and reduce tasks.

### 5.2.1 Compression Evaluation

Table 5.4 shows how the output compression influences the time values. As shown in Table 5.4, this tool produces approximately the same amount of data of the AraMorph. However, the execution time is larger than that of AraMorph. Thus, the processing time dominates the output writing time and output compression does not improve execution times.

### 5.3 Palavroso Results

This section presents the results for the Palavroso tool. Table 5.3 shows the serial execution time of Palavroso tool. The serial times of Palavroso do not measure any kind of post-processing of their results. The input and output were read and written in the local disk, respectively. Figure 5.3 shows the speedup for the inputs 1, 2, 5, 10, 20, 50, and 100 MB. Due to the very fast classification speed of this tool, the speedups greater than one only start appearing for input larger than 10 MB. This degradation for the smallest inputs is caused by the significant overhead in launching map and reduce tasks.

The last graphic in Figure 5.3 shows the speedup values for 100 MB of input text. The horizontal evolution of speedup is due to the large amount of data produced. Because the reducers are responsible
Figure 5.3: Palavroso POS Tagger speedup results
Table 5.5: Palavroso serial processing time

<table>
<thead>
<tr>
<th>Input [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>37</td>
</tr>
<tr>
<td>20</td>
<td>94</td>
</tr>
<tr>
<td>50</td>
<td>233</td>
</tr>
<tr>
<td>100</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 5.6: Palavroso output compression evaluation with a fixed number of 64 mappers and 64 reducers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>81</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>163</td>
<td>42</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>407</td>
<td>45</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>652</td>
<td>39</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>104</td>
<td>1629</td>
<td>48</td>
<td>51</td>
</tr>
<tr>
<td>50</td>
<td>278</td>
<td>4073</td>
<td>72</td>
<td>96</td>
</tr>
<tr>
<td>100</td>
<td>474</td>
<td>7820</td>
<td>106</td>
<td>144</td>
</tr>
</tbody>
</table>

A reduced number of mappers also decreases speedup. The optimum speedup value starts appearing when the number of mappers is around the number of cores.

The execution time of this tool is around 150 seconds, on the yellow portion (light gray on black-white scale) of graphs on Figure 5.1 and 350 seconds on the dark blue (black on black-white scale) portions. On the red portion (dark gray on black-white scale) the execution times are around 250 seconds.

The following section studies the impact of output compression on execution times.

5.3.1 Compression Evaluation

Due to the large amount of data produced by this tool, the compression of the final output can decrease execution time and, therefore, increase the speedup. (as shown in Table 5.6). Just like AraMorph, the compression of the final output with Gzip reduces the execution time for inputs larger than 10 MB. With compression, the output generated is, on average, 6 times larger than input, and not 80 times as in the case where no compression is used. Once again, with compression, it is possible to continue to increase
Table 5.7: Annotation merging time evaluation with a fixed number of 64 mappers and 64 reducers, for an input of 100 MB of text.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Palavroso</td>
<td>171</td>
</tr>
<tr>
<td>Palavroso + JMARv</td>
<td>323</td>
</tr>
<tr>
<td>JMARv</td>
<td>179</td>
</tr>
</tbody>
</table>

the input data and avoid scalability problems due to HDFS writing.

5.4 JMARv Results

Unlike the previous tools, JMARv does not process an input text. 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: the Palavroso execution time with output creation time, execution time of JMARv with the previously written Palavroso output and the time of Palavroso and JMARv executed in a pipeline (the 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 5.7. The first column shows the execution time of Palavroso tool. The second column shows the time of the 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 is necessary to handle about 8 GB of output produced by the last tool. However, these results show that running JMARv with this amount of data is practically the same as running both tools and only write their output at the end. This can be explained by the amount of data that is produced when running both tools in pipeline (around 12057 MB), that is around three times bigger than the one produced by the JMARv isolated execution (around 4512 MB).

5.5 Summary

This section studied the speedup achieved for a subset of the integrated tools. We showed that, in the case of computation heavy tools, it is possible to obtain high speedup values (around 40), with a higher map task number, because processing time dominates the output writing time.

In the case of the tool being fast producing its output, it is necessary to increase the number of reducers (responsible for writing the final output) in order to achieve higher speedups. However, these speedup values are lower than the ones obtained in computation heavy tools.

However, when processing previously created annotations, stored in HDFS, that were produced
by another set of tools, the processing passes from the map task to the reduce task, as explained in the annotation merging section. Our results show that running a second tool with the data (written to disk) produced by first one as input is practically the same as running both tools and only write their output at the end.

This chapter also studied the impact of compression on execution time. We showed that the computationally light tools that produce a large amount of data, compression allows the reduction of execution times, in addition to output size. However, compression of the final output raises a problem because the compressed files can no longer be processed in parallel. When a uncompressed file is processed, it is divided in a set of fragments that are assigned to different map tasks across machines in the cluster. This problem can be partially solved by creating more output files in order to reduce the size of the compressed files assigned to map tasks. If the compressed file assigned to a map task is too large, and the machine where this task is running crashes, all the computation performed by the task must be repeated in another machine. Therefore, it is important to reduce the amount of work units (file fragments) assigned to tasks.
Conclusions

The shift in hardware technology from a single core to multicore will have a profound effect in the way we do programming in the future. We’re really in the midst of a revolution in the computing industry. – Tony Hey, Microsoft Research

This framework allowed us to build scalable NLP systems that achieve significant speedups. We showed that, in the case of computation heavy tools, it is possible to obtain high speedup values (around 40), with a higher map task number, because processing time dominates the output writing time. In cases where the tool is fast producing its output, it is necessary to increase the number of reducers (responsible for writing the final output) in order to achieve higher speedups. However, these speedup values are lower than the ones obtained in computation heavy tools.

In this work, we integrated tools for four different languages. In this way, we showed that our framework supports the representation of a wide range of linguistic information varieties. When integrating these tools, it is not necessary to consider any aspect related with the parallel execution on top of the Hadoop. The programmer focuses only on representing the linguistic information produced by the tool for a given input text or a previously created annotation. On the side of parallel programming, the programming ease offered by the Hadoop framework allowed us to focus only on the problem we are solving. All the problems inherent to distributed computing are transparently solved by the platform.

The MapReduce sort/grouping capabilities allowed us to efficiently merge annotations produced by tools.

Nevertheless, the MapReduce paradigm only provides a limited vision over the key/value pairs to the map and reduce tasks. Although we have control over the assignment/grouping of pairs, it is not possible to access pairs assigned to the other tasks, since the MapReduce paradigm does not contemplate communication between tasks.

We must also consider the setup time for Hadoop. When executing the tools in parallel, 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.

The following subsections lists this work contributions and some of the improvements that can be done to this work in the future.
6.1 Contributions

The major contribution of this work is a framework that simplifies the integration of independently developed NLP tools. This framework allows the development of scalable NLP systems, that offer an easy-to-use programming environment and a transparent handling of distributed computing problems, such as fault tolerance and task scheduling. Moreover, these NLP systems are language-independent and avoid information losses between their tools.

6.2 Future Work

This section presents some items for future work, regarding both the development of the framework and its usage.

6.2.1 Framework Parametrization

Currently the NLP systems can only be constructed by using the Java API provided by the framework. However, the process can be simplified by an XML parametrization. In this way it is possible to compose a tool workflow by simply editing an XML file.

6.2.2 Hadoop Improvements

Currently, the key and value elements from the pair uses Hadoop serialization. However it is also possible to use another serialization format. One of those serialization protocol is Google Protocol Buffers (Protocol Buffers, n.d.), that was recently published. Protocol Buffers are Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data. Their representation format is smaller (3 to 10 times smaller), faster (20 to 100 times faster), and simpler than XML (Protocol Buffers, n.d.). This platform generates all source code from a definition file that contains the data to be structured. The generated source code allows an easy writing and reading of the structured data to and from a variety of data streams and using a variety of languages – Java, C++, or Python.

Besides Protocol Buffers serialization protocol, there is also the serialization representation of the Thrift (Thrift, n.d.), that is currently used to interact with existing tools, and JSON (JavaScript Object Notation) (JSON, n.d.) lightweight data-interchange format, for example. Thus it is also important to study the impact of these serialization protocols on the framework’s performance.
6.2.3 Syntactic Analysis

On the linguistic part of the work, we plan to integrate tools that produce syntactic annotations. The current representation format (Linguistic Annotation Framework) supports this kind of annotations. This linguistic information can be merged with the current tree by simply adding more nodes above the nodes that contain the morphosyntactic annotations. In this way, it is possible to represent different syntactic trees over the same morphosyntactic information.

6.2.4 Graphical Interface

The representation format chosen can make visualization difficult due to the large number of connections between graph elements in the LAF representation format and their manual editing. Hence, it is important to create a graphical environment for visualization and editing of these files. One possible solution is to adapt XMLaligner (Anjo & Matos, 2007) to handle the LAF format.

6.2.5 Tool Integration Improvements

Currently, the framework only supports four execution modes for tools: native, command line, Thrift and JNI. We plan to add more execution modes. One of those execution modes is a client for WebService based tools, where the only mode to interact with these tools is by using SOAP (Simple Object Access Protocol).

This work did not focus on information normalization. The LAF representation format does not provide specifications for annotation content categories (i.e., the contents of the associated linguistic phenomena). To solve this problem, there are several type libraries, like DCR (Data Category Registry) (ISO/TC37/SC4, n.d.), that can be explored in the future, in order to improve interoperability between NLP systems.
Bibliography


Appendices
public class StanfordEnglishPOSTagger extends Classifier {

    static {
        try {
            MaxentTagger.init(DICTATIONARY);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public StanfordEnglishPOSTagger() {
        super(new JavaExecutionMode());
    }

    public void tagg(MorphoSyntacticAnnotation a, List<String> input, List<String> output) {
        Segmentation segmentation = new Segmentation();
        MorphoSyntacticAnnotation annotation = new MorphoSyntacticAnnotation();

        try {
            long i = 0;
            String process = "";

            for (String segment : input) {
                process += segment;
            }

            List<Sentence> sentences = MaxentTagger.tokenizeText(new StringReader(process));

            for (Sentence sentence : sentences) {
                Sentence<TaggedWord> taggedSentence = MaxentTagger.tagSentence(sentence);

                for (int j = 0; j < sentence.length(); j++) {
                    TaggedWord word = taggedSentence.getHasWord(j);

                    String from = new String();
                    String to = new String();

                    from = Integer.toString(((Word) sentence.getHasWord(j)).beginPosition());
                    to = Integer.toString(((Word) sentence.getHasWord(j)).endPosition());

                    Segment s = new Segment("t" + i++, from, to, ((HasWord) word).word());
                    segmentation.addSegment(s);

                    annotation.addClassification(this.createClassification(s, word.tag()));
                }
            }

            this.addSegmentation(segmentation);
            this.addMorphoSyntacticAnnotation(annotation);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private Classification createClassification(Segment segment, String c) {
        Classification classification = new Classification();
    }
}
classification.addSegment(segment.getId());

FeatureStructure fs = new FeatureStructure();

Feature pos = new Feature();
pos.setName("pos");
pos.setValue(c);

fs.addFeature("POS", pos);

classification.addFeatureStructure(fs);

return classification;
}
Figure B.1: Architecture