Boosting Performance of a Statistical Machine Translation System Using Dynamic Parallelism

M. Fernández, Juan C. Pichel, José C. Cabaleiro, Tomás F. Pena

Centro Singular de Investigación en Tecnoloxías da Información (CITIUS)
Universidade de Santiago de Compostela, Spain

Abstract

In this work we introduce a new Statistical Machine Translation (SMT) system whose main objective is to reduce the translation times exploiting efficiently the computing power of the current processors and servers. Our system processes each individual job in parallel using different number of cores in such a way that the level of parallelism for each job changes dynamically according to the load of the translation server. In addition, the system is able to adapt to the particularities of any hardware platform used as server thanks to an autotuning module. An exhaustive performance evaluation considering different scenarios and hardware configurations demonstrates the benefits and flexibility of our proposal.

---

1. Introduction

In the modern digital society, we estimate that each day are created around 2.5 exabytes of data, in such a way that 90% of the data all over the world were created just only in the last two years [1]. Most of these data are text information written in languages we do not (fully) understand. In this way, the role of the Machine Translation (MT) in the Big Data era becomes even more relevant than years ago. However, we must take into account that an automatic translation does not have to be perfect to be useful. Depending on the use or purpose of the translation the requirements of speed and quality are different. We distinguish three categories of use of machine translation [2]: assimilation, the translation of foreign material for the purpose of understanding the content; dissemination, translating text for publication in other languages; and communication, for example the translation of emails, chats, and so on.

Nowadays the Statistical Machine Translation (SMT) dominates the field of machine translation. Companies like Google or Microsoft adopted this model for their online translation systems. SMT is an approach to machine translation that is characterized by the use of machine learning methods [3]. It is a paradigm where translations are generated on the basis of statistical models whose parameters are derived from the analysis of bilingual text (parallel) corpora and also with monolingual data. From the first ones, the system learns to translate small segments of text (translation model), and from the latter it learns how to organize the text to be fluent (language model). Once trained, an efficient search algorithm quickly finds the translation with highest probability among a large number of choices taking into account both translation and language models. In particular, considering f as the source sentence and e any of its translations into the target language, the best (most probable) translation of f is given by the following expression:

$$\hat{e} = \arg\max_{e \in E} \; p(f|e)p(e)$$

where E is the set of all sentences in the target language, $p(f|e)$ is the probability that the source sentence is the translation of the target sentence (translation model), and $p(e)$ is the probability of appearance of that target language sentence (language model). Note that the main benefits of SMT over traditional rule-based paradigms are that the engines produce more appropriate and natural sounding translations, and the technology is not customized to any specific pair of languages.

It is worth to mention that the larger the corpora used in the training of a SMT system, the better and more

---

Email addresses: `marcos.fernandez.lopez@usc.es` (M. Fernández), `juancarlos.pichel@usc.es` (Juan C. Pichel), `jc.cabaleiro@usc.es` (José C. Cabaleiro), `tf.pena@usc.es` (Tomás F. Pena)

complete translation tables and language models will be created. This leads to higher quality translations, but it comes at the cost of a significant increase in the translation times because of the greater number of translation possibilities to be evaluated. Therefore, it is important for the SMT system to make an efficient use of the hardware to extract all its computing power. In case the system accepts requests from different users, as in an online translation system, another factor that impacts the performance is the load of the translation server. Translation times will increase dramatically in case the system does not distribute the requests in a balanced way. For all these reasons it is convenient to develop solutions that take advantage of the parallelism capabilities of current computers in order to improve the overall performance of a SMT system.

In this paper we introduce a new solution for an online SMT system with the main goal of reducing the translation times exploiting efficiently the computing power of the current processors. With this objective in mind, our system processes the translation requests in parallel, translating each job using a different number of cores. We must highlight that the level of parallelism changes dynamically depending on the load of the server. This decision is also influenced by the information provided by an autotuning module, which allows our system to adapt to the particularities of the hardware platform beneath. Our translation system is based on Moses [4], which is probably the most widely used open-source implementation of the SMT paradigm. A thorough performance evaluation considering different scenarios shows the benefits and flexibility of our proposal.

Note that most of the efforts of the SMT community have been devoted to the research of various statistical methods to construct language and translation models with higher translation quality. Only few works have focused on the performance of the translation systems from a parallelism and/or load balancing perspective. To the best of our knowledge, none of those proposals present the characteristics of the SMT system introduced in this work.

The rest of the paper is organized as follows. Section 2 describes Moses focusing on some of its performance issues that our translation system should overcome. Section 3 details the architecture and operation of the new translation system. Section 4 presents the experiments carried out to evaluate the performance of our proposal. Section 5 discusses about the related work. Finally, the main conclusions derived from the work are explained in Section 6.

2. Background on Moses

Moses [4] is one of the most successful open-source implementation of the Statistical Machine Translation model. Moses consists of two main components: the training pipeline and the decoder. The training process uses as input large quantities of parallel text in such a way that each sentence in the source language is matched with its corresponding translation in the target language. Data typically needs to be preprocessed before it is used in training. Once the parallel data is ready, Moses uses occurrences of words and segments to infer translation correspondences between the two languages considered, building this way a translation model. Another important part of the system is the language model, which is a statistical model created using text in the target language and utilized afterwards by the decoder to improve the fluency of the output.

The core of Moses is the decoder, whose goal is to find the sentence in the target language with the highest score according to the translation and language models corresponding to a particular source sentence. Note that decoding is an enormous search problem, generally too big for exact search, so Moses provides different strategies to deal with this search.

Moses presents two modes of execution: Stand-alone and Server mode. In both cases the input (translation job) must be plain text and be formatted in a way that Moses can interpret it correctly. For instance, it should not contain capital letters, punctuation marks must be separated from any word by a space, etc. In the machine translation field this process is known as tokenization.

The Stand-alone mode runs directly from command line. It requires the file to translate (already tokenized) and the path to the configuration file of Moses, which contains the translation tables, language model, weights for some parameters, etc. This mode of execution admits multithreading (adding the flag `-threads`) [5]. If multithreading is enabled, Moses will use a pool of threads to translate the paragraphs (translation units/requests) in the input file.

The Server mode adds the possibility of running the translation engine as a process that listens to XML-RPC requests. XML-RPC is a remote procedure call protocol which uses XML to encode its requests and HTTP as transport mechanism. Therefore, it can attend translation requests from distributed clients written in any programming language with support for XML-RPC libraries. As the goal of our work is to develop an efficient online SMT service, we must highlight that our system is based on the operation of Moses in Server mode.

In this mode of execution, several translation jobs reaching the server at the same time are translated in a parallel way by default. However, there is a significant difference between the parallel processing used by Moses Server and Stand-alone. In particular, Moses Stand-alone automatically distributes the paragraphs (translation units) of an input file (translation job) among several threads (if the `threads` option is enabled). However, a single job is always processed sequentially in Moses Server, that is, dealing with one translation unit at a time and using only one thread. It means that a large translation job (a book, for example) will not take advantage of the parallel capabilities of the computer even when this is the only job running on the system. Consequently, Moses Server only ensures the maximum use of the computational resources when the number of simultaneous jobs sent by clients is at least equal to the number of cores available in the translation server. If we want to take advantage of the parallel processing power of the server, as we will explain in Section 3, the job must be preprocessed in order to split it up into several translation units (sentences, paragraphs, etc.) with the aim of sending them concurrently as different translation requests.

2.1. Additional limitations of Moses Server

Moses uses translation caches to store useful information that can be reused for future translations, speeding up the translation process. The way these caches are managed has changed in version 2.1 (released on January, 2014), which is the version considered in this work. Previous versions of Moses used a global cache for all the threads, so the utilization of expensive (in terms of performance) locks was mandatory to have access to it. In versions 2.1.x, Moses uses a distinct translation cache for each thread, so these locks are not needed anymore. This behavior improves the performance of Stand-alone Moses, but it affects badly to Moses Server.

As Moses uses per-thread caches, the reason of this bad behavior is related to how Moses Server handle threads. In particular, Moses Server attends each translation request using a new thread that is destroyed after completion, thus losing all the information stored in the cache. However, in Stand-alone mode a pool of threads is created in such a way that threads processing a job are always the same ones. In this way, those threads can maintain useful information in the caches and take advantage of it for each translation unit they have to process.

After several tests we have observed that, when Stand-alone mode is considered, the best performance is generally obtained using individual sentences as translation units. However, the problem detailed above about Moses Server and the thread caches entails that for versions 2.1.x, perhaps sending requests consisting of individual sentences is not the best strategy for this mode of operation. The reason is that each sentence would be translated without taking advantage of cache information, thus incurring in overhead every time.

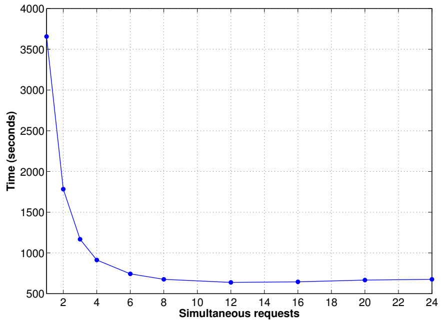


Figure 1: Moses decoding times for Spanish-English on a 24-cores machine.

Two possible solutions have been evaluated to overcome this limitation. In the first approach, we discovered that it could be found an optimal size of the translation unit so it was large enough to benefit from information stored in the caches but, at the same time, small enough to not produce memory consumption problems. Our second solution allows Moses Server to reuse the information stored in the caches, so we could use individual sentences as the translation unit. Note that the first approach yielded good results but it has also some disadvantages with respect to the second proposal. For this reason, the latter one is the solution adopted by our translation system. Details are provided in the following section.

Another important issue of Moses Server is caused by some unknown problem related to the locking mechanisms [6]. For this reason Moses is not capable of scaling when using more than 16 threads, although the scalability is already poor from 8 threads on, as Figure 1 illustrates. To avoid this restraint we could use several instances of Moses Server running on the same machine in such a way that each instance attend a maximum of 16 translation requests at a time. In this way, the translation system could scale with more than 16 threads and besides the server would support a larger workload without saturation. The counterpart is that more memory is required, so this should be considered to avoid running an excessive number of instances. In addition, as we will show later, to implement this solution it would be also necessary a new load balancer module to distribute

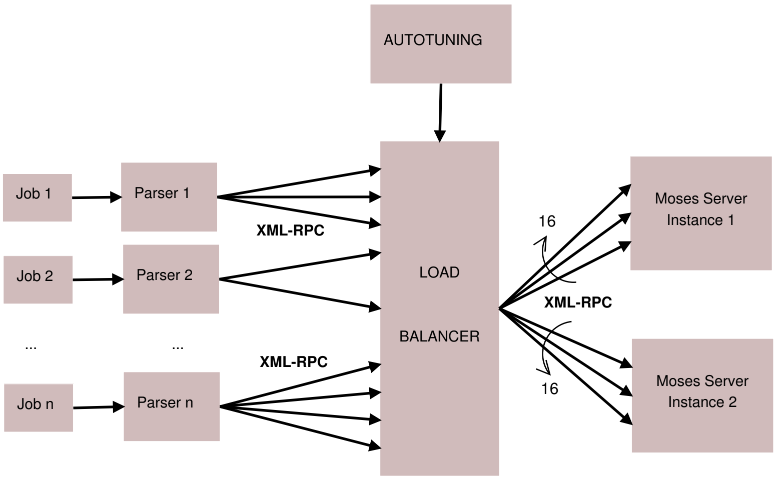


Figure 2: Architecture of the proposed translation system using two Moses Server instances.

the translation requests among the different instances.

3. Architecture of the Machine Translation System

In this section we describe the architecture of the new SMT system. As we have mentioned previously, its main goal is to decrease the translation times exploiting efficiently the parallelism capabilities of the current processors and servers. Each translation job will be processed in parallel in such a way that the level of parallelism (number of cores used) is adjusted dynamically depending on the load of the server and the particularities of the hardware used. We must highlight that the system is designed to be used as a part of a more complex infrastructure. For example, it could be the core of a complete translation web service or it could be used to carry out the translation of webpages on the fly. So these more complex systems will be the sources of our incoming translation jobs.

As noted previously, the system is based on Moses Server. However, it is worth to mention that no modifications or changes to the Moses source code are required. Therefore, our proposal allows the system to work with future (and also legacy) releases of Moses. In addition, it is also compatible with other SMT frameworks different than Moses, the only requirement is that they should accept XML-RPC requests.

A global view of the architecture of the proposed SMT system is shown in Figure 2. It consists of three modules, namely: parser, load balancer and autotuning. All of them were implemented in Python. We can summarize how the system works as follows. First, the incoming translation jobs from the clients are preprocessed by the parser (a different parser instance for each job). This preprocessing phase includes sentence splitting and tokenization. It is also responsible for the initialization of the pool of processes which will be used to send the translation requests to the load balancer. The load balancer then distributes these requests among the different instances of Moses Server that are active on the system. The autotuning module is a separate application which should be executed just one time after the installation of Moses. It provides useful information to set the appropriate level of parallelism at any given moment for the particular hardware considered.

We must highlight that these components may be running on the same server or reside in completely different machines. It means that the system has great scalability in such a way that if more computing power is needed, it is only necessary to add new hardware running more Moses Server instances. Next, a detailed description of the three modules (parser, autotuning and load balancer) is shown.

3.1. Parser module

When a translation job reaches the system it is sent to an instance of the parser module, starting the preprocessing phase. Preprocessing is quite standard and consists mainly in the tokenization of the input text and splitting the text into translation units. In the case of webpages or documents it also maintains information about how to recover the original aspect of the text after the translation procedure. Once the text is correctly preprocessed it is possible to start sending translation requests to the load balancer module.

Regarding the granularity of the translation units, the usual strategy is to split the text into individual sentences. But, as it was stated in Section 2.1, maybe using individual sentences is not the best strategy for Moses Server in case the information in the translation caches cannot be reused among requests. As it is explained in Section 3.3, we found a way in which Moses Server can make use of this information instead of discarding it after a translation request is completed. In this way, translation units in our translation system are always individual sentences of the text.

The parser module sends requests to the load balancer using XML-RPC. The usual way to make these requests is serially. But, what would happen if a large document is sent to translation when the load of the server is low? In that case, we would be wasting computing power because most of the processors of the server would be idle, waiting for new jobs to process. In this way, the client would not get the best possible response time. The solution is to send several translation requests (of the same job) simultaneously. In other words, the translation of the document will be performed using various processors (cores) at the same time.

To attain this goal the parser creates a pool of processes which iterates over a list containing all the tokenized translation units of the input text, sending a translation request per unit in that list. Thus, the number of processes of the pool will determine the maximum number of simultaneous requests belonging to that job that could be processed in parallel by Moses. The optimal number of simultaneous requests at any given time is automatically provided by the load balancer module, as we will further see. Once the pool of processes is created, it starts sending parallel XML-RPC requests to the load balancer until all the translation units of the job are returned correctly translated. Finally, the parser module will recover the original aspect of the document.

3.2. *Autotuning module*

This module is an independent application executed only once after the installation of Moses. Its execution can last up to several hours, depending mainly on the speed and number of processors of the nodes used as server. The mission of this module is to define the different levels of parallelism (number of cores) that the system will use depending on the incoming rate of translation jobs and the size of the input text. It means that the number of cores used in the translation of a small document and a book, under the same load conditions in the server, could be different. It is worth to mention that the load balancer module will be the responsible for measuring the server load and dynamically adjust the degree of parallelism accordingly.

The output of the autotuning module consists of several files with information regarding the permitted levels of parallelism to process the input text of a particular size. In all the experiments shown in this paper we have classified the text size into four categories based on our experience:

- *xsmall*: less than 400 words
- *small*: 400 – 1,300 words
- *medium*: 1,300 – 5,000 words
- *large*: more than 5,000 words

In this way, the autotuning module will generate four files. We must highlight that our system is flexible enough to allow any categorization of the input text sizes.

```
 1  1
 2  1
 3  1 *
 4  1

xsmall-levels
```

```
 1  1  8.4
 2  1  4.5
 3  1  3.8
 4  1  3.2
 6  1  3.4
 8  1  3.5
12  1  4.2

xsmall-times
```

(a)

```
 1  1
 2  1
 3  1
 4  1
 6  1 *
 8  1
12  1

large-levels
```

```
 1  1  436.8
 2  1  220.1
 3  1  157.8
 4  1  125.4
 6  1   94.1
 8  1   76.6
12  1   55.5

large-times
```

(b)

Figure 3: Files generated by the autotuning module for very small (a) and large (b) input texts on the `ctserv01` system.

Examples of output files for very small and large text sizes are shown in Figure 3 (labeled as `levels`). To obtain these values the autotuning module was executed on a 12-core system (`ctserv01`, see Section 4.1 for details). Each line of the files contains a permitted configuration defined as a pair *level of parallelism – optimal size of the translation unit*, sorted from lower to higher level of parallelism. The load balancer module will select one of these configurations to process the incoming jobs depending on the load of the server at a specific time. In particular, the first digit of each line indicates the number of translation requests from an individual job that will be sent simultaneously to Moses Server (level of parallelism). That is, the number of cores that will process the job in parallel. In this way, in the examples of Figure 3, very small texts could be translated using from 1 to 4 cores (`xsmall-levels`), while large documents could be processed using up to 12 cores at the same time (`large-levels`).

In the examples of Figure 3, the size of the optimal translation unit (expressed in number of words) is always 1 for all the levels of parallelism. Note that a sentence will never be divided into smaller units, so size 1 means that the translation unit size is equal to one sentence. As noted previously, this is the default unit size used by our SMT system.

On the other hand, we have also considered a different solution that takes advantage of larger translation units to alleviate the thread caches issue explained in Section 2.1, and thus it requires to find the optimal sizes. Although this is no longer needed because its performance is lower than the solution that reuses the information of the thread caches, the autotuning module maintains the ability to find the optimal translation unit size if it was necessary.

The "*" symbol beside one of the configurations means that it is the current configuration selected by the load balancer module for a particular input size. In this way, considering the example of Figure 3a, if a very small text job just arrives to the translation system, it will be assigned to a translation pool of 3 processes. In other words, the maximum number of sentences belonging to this job being translated concurrently by Moses Server is 3.

In order to generate the configuration files, the autotuning module uses a testbed which contains hundreds of input texts of different sizes. It calculates the average translation time for each text size category (that is, xs-mall, small, medium and large texts in our case) using a set of predefined configuration pairs *level of parallelism - translation unit size*. The number of configurations evaluated by the autotuning module depends on the number of available processing cores in the system. Figure 3 shows the configuration pairs evaluated and its corresponding average translation time for very small and large input texts on the `ctserv01` system (`xsmall-times` and `large-times`, respectively).

For each text size category, a configuration pair is included in its corresponding `levels` file only if the average translation time using this configuration is at least 10% lower than the obtained by the selected configuration of the preceding level. If none of the configurations for a particular level fulfill that condition, the preceding level is considered the maximum level permitted. This is done because increasing the level of parallelism implies using more resources (cores), but this is not worthy if there is not a perceptible improvement in the performance. We illustrate this behavior using the example of Figure 3a. In this case the average translation time using 4 and 6 cores (`xsmall-times` file) does not reach the 10% threshold. In fact, translation times do not scale using more than 4 cores. Therefore, the maximum level of parallelism for very small texts will be 4 (see `xsmall-levels` file). On the other hand, only the best configuration pair for each level of parallelism can be selected to be part of the `levels` files.

3.3. Load balancer module

The load balancer module is a XML-RPC server and it is responsible for:

- Distributing the translation requests among the different instances of Moses Server.
- Monitoring the system load.
- Modifying dynamically the level of parallelism.

The communication with the Moses Server instances is made through the XML-RPC protocol. The first task to make XML-RPC requests is to initialize a connection object that, among other information, contains the address and port where the server is listening. Then the remote method is called using that object. These connection objects cannot be used simultaneously by two requests, so the usual way of making a XML-RPC call is to initialize a new object each time a remote call is made. This was our first approach, and it resulted in the aforementioned overhead because the information in the translation caches cannot be reused between requests.

To overcome that limitation we used a different approach. Instead of creating a new object for each request, a pool of connection objects is created. In this way, each object in the pool is used sequentially, but the global operation will be performed in parallel. The idea behind this strategy is to be able to reuse the same connection objects among requests, instead of creating a new object for each one. This implementation shows a good behavior as it takes profit of the information stored in the translation caches while that connection is open.

As stated in Section 2.1, Moses does not scale beyond 16 threads, so a pool of 16 connection objects per instance is enough. This pool is created, initialized and updated by the load balancer module. It also maintains the state information about each connection object (busy or free). In order to distribute the requests among the different instances the load balancer checks if there is an object available for each translation request that reaches the system. If this is the case, it sends the request using that object. If no connection objects are available, the load balancer puts that request on hold for

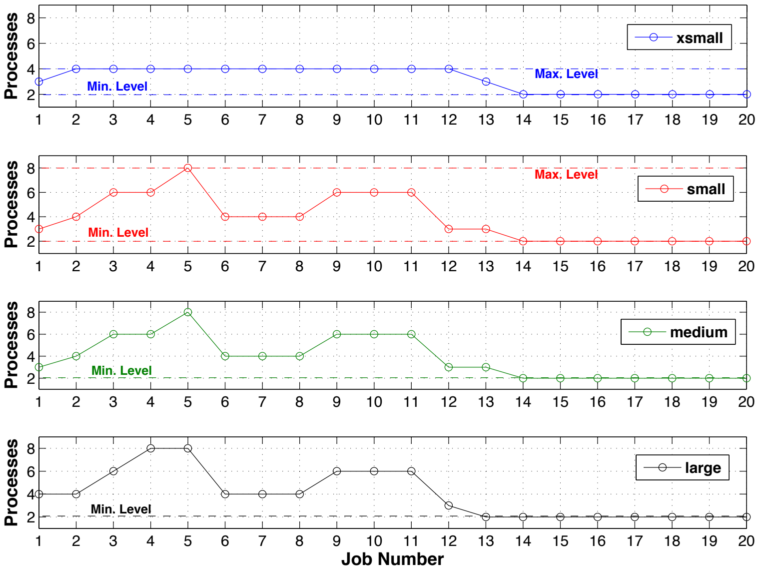


Figure 4: Example of how the SMT system changes dynamically the level of parallelism assigned to each job request according to the load on the server.

some random amount of time, trying again until eventually the request is successfully sent to an instance of Moses Server. When the load balancer receives the response, it is redirected to the parser so the original text can be reconstructed.

The other two functionalities of the load balancer module, monitoring the server load and adjusting the level of parallelism, are interrelated tasks. A high level of parallelism is beneficial in situations where the load of the server is low, but it could provoke saturation if a high number of translation jobs are being received. And vice versa, a low level of parallelism would be desirable when the load is high, but it would imply wasting computing resources in the opposite situation.

To attain this goal of dynamically adjusting the level of parallelism depending on the server load, the load balancer keeps count of all the requests it receives for a certain period of time. With this information and using some predefined thresholds, the load balancer decides if the level of parallelism should be increased, maintained or decreased. Only the configurations determined by the autotuning module (included in the `levels` files) can be selected. In this way, the load balancer must take a decision regarding the level of parallelism periodically. It may be noteworthy that if the level is modified, it only applies to the new jobs reaching the system. Those jobs that are already being translated maintain the level previously assigned to them.

Therefore, the level of parallelism will fluctuate dynamically depending upon the load of the server in such a way that more resources will be assigned to jobs that reach the server in a moment of low load with respect to jobs arriving to a high loaded server.

3.4. Example of the translation system operation

Next we will illustrate how the translation system works. In this example one job is sent every 20 seconds to the system (`ctserv01` server, see Section 4.1 for details). The size of these jobs alternates cyclically using the sequence: very small, small, medium and large. It means that the first job is very small, next one is small and arrives 20 seconds later, and so on. In the end 30 jobs of each size are sent to the system for a total of 120 jobs. Figure 4 shows the level of parallelism assigned by the load balancer to each job and how it changes dynamically through time according to the load of the server. Only the first 20 jobs for each size are represented in the graphics because the situation remains stationary until job 30.

The autotuning module resolved that, for this test server and configuration, a translation pool of more than 4 processes is never beneficial for very small jobs. In other words, the maximum level of parallelism allowed for very small jobs is 4 ("Max. Level" line in the figure). For the same reason, small jobs should never exceed 8 processes, while medium and large jobs could potentially use up to 12 processes. Note that in this example the highest level of parallelism reached is 8 for some jobs of small, medium and large sizes. We found that generally it was better in terms of performance not allowing serial processing, so two processes is the minimum level of parallelism permitted.

At the beginning the system was idle. By default, translation pools are initialized with 3 processes (see Job #1 of xsmall, small and medium sizes). However, the level of parallelism of the first large job increases between the arrivals of the third job (Job #1 - medium, $t = 40$ seconds) and the fourth one (Job #1 - large, $t = 60$ seconds). Afterwards we observe that the level increases quickly because the rate of incoming jobs is not very high. At some point the number of concurrent requests being processed by the system (that is, the load of the server) is high. This is caused by the higher number of simultaneous requests sent to the system per job (level of parallelism), and also by some medium and large jobs whose translations are unfinished (some of them last several minutes to complete). As a result the load balancer detects that the maximum load threshold has been reached and it decides that the level of parallelism should be reduced. We must highlight that increments are done in steps of one level and decrements in steps of two levels, as experimentally was determined to be the best strategy. This is a conservative strategy which tries to avoid the saturation of the system as soon as any evidence of high load in the server is detected by the load balancer. The level of parallelism keeps fluctuating according to the guidelines of the load balancer until it reaches a stationary state. Note that, depending on the incoming rate of jobs, the system could reach a totally different stationary state. Of course it is also possible that the levels change dynamically during the entire test with no stationary states.

4. Performance Evaluation

In this section we will show the performance results obtained using our SMT system.

4.1. Configuration

The translation system was tested on two different hardware platforms:

- Server `ctserv01`: It consists of 2 CPUs Intel Xeon E5-2630L at 2.4 GHz (2×6 cores, Ivy Bridge microarchitecture), 32 GB RAM, and Hyper-threading disabled.
- Server `ctserv02`: It consists of 2 CPUs Intel Xeon E5-2650L at 1.8 GHz (2×8 cores, Sandy Bridge microarchitecture), 64 GB RAM and Hyper-threading enabled.

For these tests all the modules (parser, load balancer and Moses instances) reside in the same machine. The translation system is based on Moses 2.1.1. We used a binarized language model and the compact representation for phrase and reordering tables, resulting in a total size for all the models of 4.5 GB. Models are for the Spanish-English pair, which is our only translation direction in all the tests. The system was trained using corpora from the European Union documentation, European Parliament Proceedings and other international organization and universities. In particular, 217 million words in English and 243 million words in Spanish were used.

Text Size	**Sentences**	**Words per sentence**	**Size (KB)**
xsmall	10.5	23.1	1.4
small	45.4	16.6	4.7
medium	164.8	13.6	13.8
large	809.7	11.2	56.8

Table 1: Characteristics of the input texts used in the performance evaluation (average values).

Transparent huge pages are enabled on both servers, as recommended in the Moses documentation. In this way, the operating system will always attempt to satisfy a memory allocation using huge pages (2 MB). If no huge pages are available (due to non availability of physically continuous memory, for example) the kernel will fall back to the regular page size (4 KB).

Extracts of different sizes from some well-known books in Spanish were used as input texts in our experiments. In particular, the dataset consists of 120 texts whose main characteristics are summarized in Table 1.

4.2. Methodology

Two case studies were considered: a scenario where a server is constantly receiving translation requests, and another in which the server is idle for a certain period of time (it could happen at night, for example) and it receives a single request. In more detail:

- *Case A:* Jobs are sent periodically to the translation server, starting from a very small job, then a small one, medium, large, and start over again until we have sent 120 texts. The time interval between jobs can be shorter or longer depending on the level of stress we want to simulate. Three types of stress (low, medium and high) were studied. For simulating a low stressed server a job is sent every 30 seconds, for medium stress every 20 seconds, and finally for high stress, jobs arrive at the system every 10 seconds. It could be argued that an actual translation server could receive translation requests at a rate higher than one request each 10 seconds, but it must be considered that half of the translation requests are of several pages size (even dozens for the larger jobs), which are way bigger than a typical translation request. So this rate of incoming requests ensures a high occupancy of the server.

- *Case B:* This case simulates when a single isolated translation job is received by the translation server. In this scenario our system would evolve to a state where the maximum level of parallelism is selected for all the incoming jobs. For this reason all the texts in the dataset are translated using the maximum level of parallelism allowed for each job size. We compare these translation times with those obtained by the serial translation of individual jobs used by Moses.

In both cases we will show results considering different number of instances of Moses Server to demonstrate the improvements achievable by using more than one instance for each translation direction.

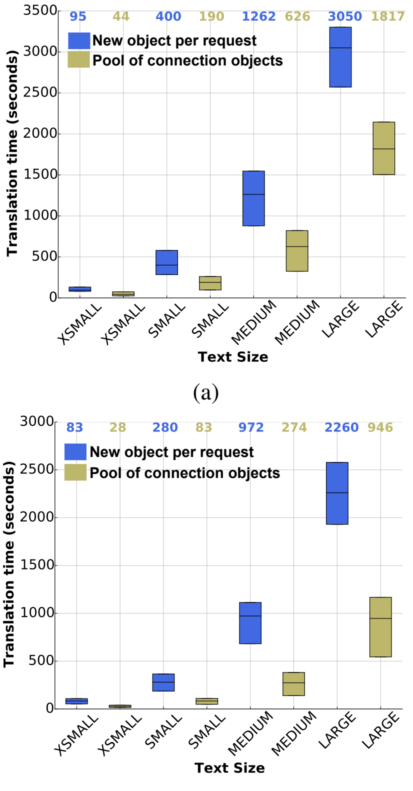
In order to illustrate the performance results in terms of translation times we present some boxplot graphs, where the top and bottom of the boxes represent the third and first quartile of the obtained results respectively. The line that crosses the boxes is the median time, whose numeric value is displayed at the top of the graph for an easier comparison. Note that considering only one estimator (such as the average execution time, for example) is not the best choice to compare the performance results because of the variability in the measurements. Boxes provide a better idea of the overall performance of the system.

4.3. Reusing the information of translation caches

Before analyzing the two case studies commented above, we will focus on showing the differences in terms of performance between using a new connection object per translation request and using a pool of preexisting connection objects.

Moses Server (by default) is not capable of reusing the information stored in the translation caches between requests. Using larger translation units could alleviate this lack of cache information reuse but, as we show next, it is not the best option. As explained in Section 3.3, we found a solution by initializing a pool of preexisting connection objects and sending the translation requests through them instead of creating a new connection object for each request.

Figure 5 shows the difference between using a new connection object per translation request and using a pool of connection objects. In particular, these graphics correspond to a situation of high stress using different number of Moses instances to attend the translation requests on the `ctserv01` system.



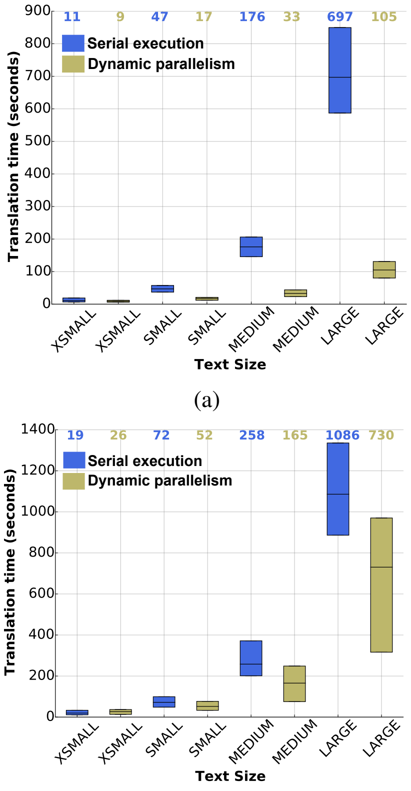
(a)

(b)

Figure 5: Translation times using a new connection object per translation request and a pool of connection objects. Measurements were performed running one (a) and two (b) Moses instances on `ctserv01`.

The size of the translation units used for this test is the most beneficial for each case. It means that, when a new connection object per request is created, translation units are portions of text of the optimal granularity calculated by the autotuning module. This corresponds to our first approach, explained in Section 2.1, with the aim of mitigating the effect of not reusing the translation caches. However, once the cache information can be reused by introducing the pool of connection objects, individual sentences become again the best translation unit size. As both figures show, the benefits of using the pool of connection objects are evident for all text sizes, with improvements superior to 2× in most of the cases. When using two instances the difference is even more noticeable, with improvements greater than 3×.

It can also be observed the improvement that comes from using several instances of Moses Server to perform the translation, even residing in the same machine. Approximately, a doubling of the performance can be observed when using the pool of connection objects. It proves to be a successful way of avoiding, or at least alleviating, the problems that Moses has with the locking



(a)

(b)

Figure 6: Translation times processing each job serially (Moses default) and using dynamic parallelism under different load conditions on `ctserv01` when running one instance: low (a) and medium (b) stress.

mechanism.

From now on, all the performance results were obtained making use of the pool of preexisting connection objects and, consequently, the translation unit will always be an individual sentence.

4.4. *Case A: experimental results*

Next, a comparison between our translation system, which has the capability of translating in parallel both a single job and multiple jobs, and the default Moses strategy where concurrency only affects to multiple incoming jobs is shown. We also demonstrate the benefits of our dynamic parallelism strategy against a fixed parallelism approach.

4.4.1. *Dynamic parallelism vs. serial execution of individual jobs*

As we have stated previously, Moses Server processes sequentially each individual job but it has the capability of performing concurrently the translation of several job requests. However, our system exploits the parallelism of a server in two levels. First, processing a single job in parallel using different number of cores, and second, allowing the concurrent translation of several jobs in the system. A comparison in terms of performance between our proposal and the usual Moses strategy is shown.

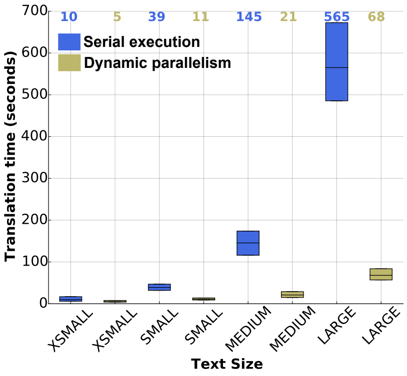
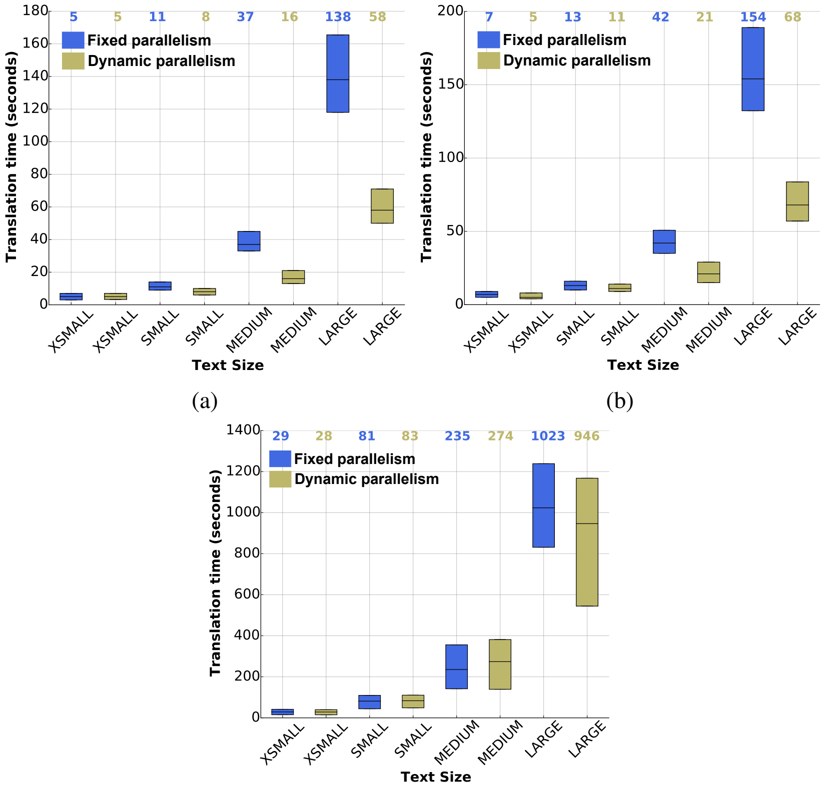


Figure 7: Translation times processing each job serially (Moses default) and using dynamic parallelism under medium stress load conditions on `ctserv01` when running two instances.

An example considering different stress conditions on the `ctserv01` platform is displayed in Figure 6. First, we focus on a situation of low stress in such a way that one translation job reaches the server every 30 seconds (see Figure 6a). If Moses processes each job sequentially, all the cores will be busy only if the same number of jobs are running on the system. Therefore, considering `ctserv01`, a minimum of 12 jobs are required to occupy all the cores available in the system. As a consequence there is an important waste of computing power for the first incoming jobs as many cores are unused waiting to process new jobs. On the other hand, with this rate of incoming jobs, many of the smaller texts get translated before the next job arrives at the system, releasing the resources which attended those jobs. Therefore, more jobs than cores are necessary to fill the system when they are processed sequentially.

However, the waste of computing power is reduced to the minimum when each individual job is processed in parallel. For example, let us consider that the translation system determines that three parallel requests per job is the initial configuration for the levels of parallelism. After only four jobs reach the system, up to 12 cores could be executing translation tasks, occupying all the resources available. As explained above, some jobs might have finished before the fourth job arrives, so the occupation would be actually lower, but the difference is still obvious.

Results in Figure 6a reflect noticeable improvements



(a)

(b)

(c)

Figure 8: Translation times considering fixed and dynamic parallelism using two instances on `ctserv01` under different load conditions: low (a), medium (b) and high (c) stress.

for all text sizes when using our approach. For instance, speedups up to 7× for the larger jobs are achieved. In this example, the autotuning module decided that for smaller texts more than 4 parallel requests were not beneficial, while for medium and large texts up to 12 parallel requests could be used if the system evolves to that level of parallelism (see Figure 3). It means that when a similar number of small and large texts are reaching the system, smaller jobs are slightly penalized because their chances of getting a free connection object from the pool to send a translation request are lower.

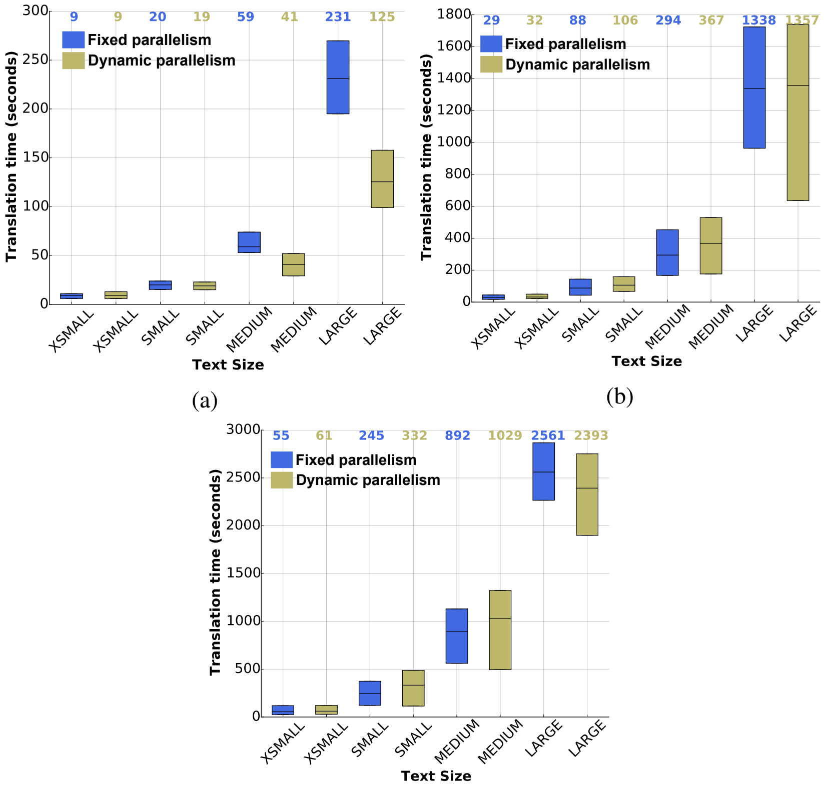
Figure 6b shows the same comparison but under a situation of medium stress. Here we can see that our system decrease significantly the translation times for all text sizes except for the smaller ones. As expected, improvements are not as good as in a low stress scenario as the higher rate of incoming jobs ensures a better use of computational resources in the case of serial execution.

Both strategies tend to converge for a high stress scenario. In that case, the rate of jobs arriving to the system is enough to maintain all the resources busy even with serial processing of each job. On the other hand, our system will gradually decrease the level of parallelism until it reaches the minimum. Therefore, if enough time goes by, both strategies will basically behave in the same way.

Finally, Figure 7 shows the same situation of medium stress than in Figure 6b but using two instances of Moses Server instead. Moses does not scale very well from 8 threads on (see Figure 1). Using more than one instance greatly improves the performance as each instance will enter the poor scalability zone less frequently. It must be noted that for the system using serial processing for each job, two Moses Server instances approximately duplicates performance. However, our system gets speedups higher than 5× for all the cases considered. This behavior is due to the static nature of the sequential system which does not have the flexibility to increase the number of simultaneous translation requests to avoid wasting computing power.

4.4.2. Dynamic parallelism vs. fixed parallelism

Until now, we have demonstrated that processing individual jobs serially leads to a waste of computational



(a) (b)

(c)

Figure 9: Translation times considering fixed and dynamic parallelism using two instances on `ctserv02` under different load conditions: low (a), medium (b) and high (c) stress.

resources when the rate of incoming jobs is not enough to completely occupy the translation server. Our proposed system solves this issue by dynamically changing the level of parallelism depending on the server load. But we wanted to check if a simpler strategy with a fixed level of parallelism could lead to similar results. After some experimentation we determined that assigning always a pool of four processes to each incoming job would be a good compromise between load and speed for our test servers. So now we will show some results comparing both strategies.

Figure 8 displays the performance of both approaches using two instances of Moses Server on `ctserv01` under situations of low, medium and high stress. Considering two instances, this test server is capable of attending much more requests than those generated in low and medium stress scenarios. So in these two situations, our system clearly outperforms the fixed parallelism strategy by elevating the degree of parallelism and, as a consequence, it exploits efficiently the computational resources.

In the high stress situation, however, the fixed parallelism strategy generates sufficient translation requests to keep the translation server busy. In this way, the performance is comparable to the dynamic parallelism strategy, which never uses the highest levels of parallelism. In particular, our system evolves to a state of minimum parallelism where two processes are used to handle each job (serial processing was discarded, as explained in Section 3.4).

Figure 9 shows the same comparison between fixed and dynamic parallelism but on the `ctserv02` server. This server consists of processors with a different microarchitecture and lower clock frequency with respect to the ones installed in the `ctserv01` system, which means it is a slower performer. As a consequence less simultaneous requests are needed to completely occupy its processing resources. If, in the previous case, our translation system clearly outperformed the fixed parallelism strategy in situations of low and medium stress, here it can be observed how a situation of medium stress is enough to completely load the system and both strategies show a similar performance.

A question that could arise is why not use a higher

Text Size	**Serial**	**Parallel, one instance**		**Parallel, two instances**	
		Time	Speedup	Time	Speedup
xsmall	8.5	4.1 (4)	2.1×	3.4 (4)	2.5×
small	31.7	7.9 (8)	4.0×	7.2 (8)	4.4×
medium	121.0	22.2 (12)	5.5×	16.1 (12)	7.5×
large	454.5	82.1 (12)	5.5×	54.4 (12)	8.4×

Table 2: Average translation times (in seconds) of one single job on the `ctserv01` system.

Text Size	**Serial**	**Parallel, one instance**		**Parallel, four instances**	
		Time	Speedup	Time	Speedup
xsmall	14.9	5.1 (4)	2.9×	5.5 (4)	2.7×
small	56.7	11.9 (8)	4.8×	9.8 (8)	5.8×
medium	213.4	37.2 (12)	5.7×	19.8 (16)	10.8×
large	819.5	139.2 (12)	5.9×	65.04 (16)	12.6×

Table 3: Average translation times (in seconds) of one single job on the `ctserv02` system.

level of fixed parallelism, with 8 or 12 processes for example. The reason is that, on the one hand, it could potentially cause saturation problems by an excessive memory consumption of the load balancer or even network congestion if the parser and the load balancer reside in different servers. On the other hand, as our experiments supported, using more processes only would help during low load situations, while for medium and high load scenarios we would observe a very important degradation in the performance.

4.5. Case B: experimental results

For this scenario a summary of the performance results obtained is shown in Tables 2 and 3. For comparison purposes those tables include the average translation times achieved by a system which uses the default Moses serial processing for each translation job and also by our system. All times are expressed in seconds. Between brackets it is also displayed the number of processes used in the initialization of the translation pool for each job. This number corresponds to the maximum level of parallelism indicated by the autotuning module for that particular text size.

Results for our first test server using one and two Moses Server instances are shown in Table 2. In this system there is enough memory to easily run more instances, but as the number of processing cores available is not really high (12), running more than two instances does not suppose a noticeable improvement in scalability. For the second server (Table 3), one and four instances were considered. In this case the number of simultaneous processes is 32 (16 physical cores, 32 threads with hyper-threading enabled) so it greatly benefits of a higher number of instances.

Both tables confirm the good behavior of our solution with respect to the serial implementation. In this way, users of the translation system will get a much faster response time for all the text sizes when the load of the server is minimum. Note that speedups are never lower than 2×, reaching values up to 12×. The sequential system could also benefit of using more than one instance, but it should implement a way of load balancing as our system does.

5. Related Work

Machine Translation (MT) is a subfield of computational linguistics that investigates the use of software applications to translate text from a source language to another target language. There are two main types of machine translation to consider, attending to its core methodology: Rule-Based Machine Translation (RBMT) and Statistical Machine Translation (SMT).

Rule-based Machine Translation uses linguistic rules to analyze the input text content in the source language to generate text in the target language. This process requires extensive lexicons with morphological, syntactic, and semantic information, and large sets of rules. The software uses these complex rule sets and then transfers the grammatical structure of the source language into the target language. These rules must be carefully designed and implemented by human experts.

RBMT is specially suitable for building online dictionaries, as its output is consistent and predictable. It usually also works well for translations between closely related languages. GramTrans [7] and Apertium [8] are two examples of machine translation platforms which use this model.

On the other hand, Statistical Machine Translation is characterized by the use of machine learning methods. It generates translations using statistical translation models obtained from the analysis of both bilingual and monolingual text corpora. From these data it automatically learns to translate small segments of text and also to organize them in a way that is fluent in the target language. As we have mentioned previously, the main advantage of SMT over traditional RBMT methods is that more appropriate and natural sounding translations are produced by the translation engines. In addition, the technology is not customized to any specific pair of languages and training is automated and cheaper when the desired corpora exist and it is good.

Our translation system is based on Moses [4], which is probably the most important open-source toolkit for SMT, but there are other relevant SMT tools such as Jane [9], UCAM-SMT [10], Phrasal [11] and Joshua [12], among others. In addition, some of the most well-known machine translation web services, as Google Translate [13] and Microsoft's Bing Translator [14], use the statistical approach in their platforms [15, 16].

However, SMT is not exempt of drawbacks as parallel corpora of good quality are not always available. Besides, it also has high CPU, disk space and memory requirements to build and manage large translation models. Precisely, the fact that SMT techniques are very CPU intensive and time consuming make them very good candidates to take advantage of parallel computing techniques for increasing their performance. However, most of the research in the SMT field has been devoted to obtain language and translation models with higher quality instead of focusing on the performance of the translation systems from a parallelism and/or load balancing point of view. In any case, we can find in the literature some examples of the latter category of works. For instance, in [5] the author describes the extension of Moses to support multi-threaded decoding. Chen *et al.* [17] show how to parallelize a MT decoder using a method called functional task parallelism, which tries to overcome some limitations posed by traditional thread-based methods. Some researchers try to exploit the massive parallelism of GPUs in order to boost the performance of the machine translation process and other natural language processing applications [18, 19]. Some implementations are based on the Map-Reduce paradigm, but they deal with the stages of training and the construction of the statistical model [20, 21]. In a more recent work, authors use Big Data technologies to process huge amounts of text using several natural language modules [22], but machine translation is not considered in the paper.

Note that most of the works commented above change the decoder or other fundamental parts of the translation system, creating an *ad hoc* implementation for a particular parallel architecture. However, our approach improves the performance of Moses without applying any kind of modification to the original Moses source code. In this way, we assure the compatibility of our solution to any release of Moses (future or legacy).

Finally, ScaleMT [23], MT Server Land [24] and MT-Monkey [25] are infrastructures for machine translation that are similar in concept to the approach explained in this paper. However, they lack the fundamental feature of allowing the parallel translation of single jobs, which permits to take advantage of all the computational resources of a server even in situations of low load. In addition, unlike these solutions, our system is able to adapt to the particularities of any hardware platform thanks to the autotuning module.

6. Conclusions

We have developed a new Statistical Machine Translation (SMT) system based on Moses that efficiently exploits the computational resources of modern servers. In addition, it is able to adapt to the particularities of the considered hardware platform and to the rate of incoming jobs. The capability of processing a single job in parallel allows our system to be much faster than other machine translation services in scenarios with few clients generating translation jobs. Besides, the dynamic nature of our system ensures that the computing power is not underused in those situations and, at the same time, minimizes memory consumption and network usage when the system is heavily loaded. It is also easily scalable thanks to its modular conception, so performance can be increased without difficulty just by adding new Moses server instances. An exhaustive performance evaluation considering different scenarios has demonstrated the benefits and flexibility of our proposal.

Our solution also avoids or mitigates some of the shortcomings that we encountered in Moses. First, by using the load balancer and several instances to perform the translations we can circumvent to some extent the locking problems which produce bad scalability from certain number of threads on. And second, introducing the pool of connection objects we also solve the problem which did not allow to take advantage of the information stored in the translation caches among different translation requests.

Acknowledgments

This work was part of the CELTIC research project (ITC-20113031) and it was developed in collaboration with **Imaxin**|Software. Authors are also supported by MINECO (Spain) grants TIN2013-41129-P and TIN2014-54565-JIN.

References

[1] IBM, Big Data at the speed of business, `http://www-01.ibm.com/software/data/bigdata`, accessed April 6, 2015.

[2] P. Koehn, Statistical Machine Translation, 1st Edition, Cambridge University Press, New York, NY, USA, 2010.

[3] A. Lopez, Statistical machine translation, ACM Computing Surveys 40 (3) (2008) 8:1–8:49.

[4] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, E. Herbst, Moses: Open source toolkit for statistical machine translation, in: Proc. of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions, 2007, pp. 177–180.

[5] B. Haddow, Adding multi-threaded decoding to Moses, Prague Bulletin of Mathematical Linguistics 93 (2010) 57–66.

[6] Moses Support Forum, Multithreading is broken for hierarchical Moses, `https://github.com/moses-smt/mosesdecoder/issues/39`, accessed February 9, 2015.

[7] GrammarSoft ApS and Kaldera Språkteknologi AS, GramTrans, `http://gramtrans.com`, accessed April 6, 2015.

[8] Apertium, `https://www.apertium.org`, accessed April 6, 2015.

[9] J. Wuebker, M. Huck, S. Peitz, M. Nuhn, M. Freitag, J. Peter, S. Mansour, H. Ney, Jane 2: Open source phrase-based and hierarchical statistical machine translation, in: International Conference on Computational Linguistics (CoLing), 2012, pp. 483–491.

[10] University of Cambridge, UCAM-SMT: the Cambridge statistical machine translation system, `http://ucam-smt.github.io`, accessed April 6, 2015.

[11] S. Green, D. Cer, C. D. Manning, Phrasal: A toolkit for new directions in statistical machine translation, in: Proc. of the 9th Workshop on Statistical Machine Translation, 2014, pp. 114–121.

[12] Johns Hopkins University, Joshua machine translation toolkit, `http://joshua-decoder.org`, accessed April 6, 2015.

[13] Google, Google Translate, `https://translate.google.com`, accessed April 6, 2015.

[14] Microsoft, Bing Translator, `http://www.bing.com/translator`, accessed April 6, 2015.

[15] Google, Machine translation - Research at Google, `http://research.google.com/pubs/MachineTranslation.html`, accessed April 6, 2015.

[16] Microsoft, Automatic translation and Microsoft Translator, `http://www.microsoft.com/translator/automatic-translation.aspx`, accessed April 6, 2015.

[17] L. Chen, W. Huo, H. Mi, Z. Zhang, X. Feng, Z. Li, Parallelizing a machine translation decoder for multicore computer, in: Proc. of the 7th Int. Conference on Natural Computation, 2011, pp. 2220–2225.

[18] C.-Y. Lai, Efficient parallelization of natural language applications using GPUs, Master's thesis, EECS Department, University of California, Berkeley (2012).

[19] S. Gupta, M. R. Babu, Generating performance analysis of GPU compared to single-core and multi-core CPU for natural language applications, International Journal of Advanced Computer Science and Applications (IJACSA) 2 (2011) 50–53.

[20] C. Dyer, A. Cordova, A. Mont, J. Lin, Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce, in: Proc. of the 3rd Workshop on Statistical Machine Translation, 2008, pp. 199–207.

[21] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, K. Olukotun, Map-Reduce for machine learning on multicore, in: Neural Information Processing Systems (NIPS), MIT Press, 2006, pp. 281–288.

[22] R. Agerri, X. Artola, Z. Beloki, G. Rigau, A. Soroa, Big data for natural language processing: A streaming approach, Knowledge-Based Systems 79 (2015) 36–42.

[23] V. M. Sánchez-Cartagena, J. A. Pérez-Ortiz, ScaleMT: a free/open-source framework for building scalable machine translation web services, Prague Bulletin of Mathematical Linguistics 93 (2010) 97–106.

[24] C. Federmann, A. Eisele, MT Server Land: an open-source MT architecture, Prague Bulletin of Mathematical Linguistics 94 (2010) 57–66.

[25] A. Tamchyna, O. Dušek, R. Rosa, P. Pecina, MTMonkey: A scalable infrastructure for a machine translation web service, Prague Bulletin of Mathematical Linguistics 100 (2013) 31–40.