REMEDI

Robust and Efficient Machine Translation in a Distributed Infrastructure

Bi-Annual Report: Month 24

Principal Investigator:

Dr. Christof Monz
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

Phone: +39 (0)20 525 8676
Fax: +39 (0)20 525 7940
E-mail: c.monz@uva.nl

Scientific Programmer:

Dr. Ivan S. Zapreev
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

E-mail: i.zapreev@uva.nl

Month	Deliverable	Progress
6	Implementation of language model data structure; implementation of multi-threading with large shared data structures; month 6 report	✓
12	Implementation of translation and reordering model data structures; implementation of decoder search and pruning strategies; month 12 report	✓
18	Implementation of search lattice data structure and feature passing; implementation of server front end; implementation of robustness preserving load balancing and restarting; complete software deliverable of decoder infrastructure; month 18 report	✓
24	Implementation of language model server in distributed environment; implementation of distributed translation model service; month 24 report	✓
30	Implementation of distributed reordering model services; complete software deliverable of distributed translation infrastructure; Implementation of k-best hypothesis extraction; month 30 report	
36	Implementation of Margin Infused Relaxed Algorithm for parameter tuning; implementation of distributed optimization infrastructure, including hyper-parameter estimation; final report; complete manual	

Contents

1 Introduction **3**

2 Deliverables **4**

3 Delivery Details **5**
3.1 Communication protocols . . . 5
3.1.1 Common base class . . . 6
3.1.2 (SL) - Supported languages . . . 8
3.1.3 (T) - Translation . . . 9
3.1.4 (PP) - Pre/Post processing . . . 12
3.2 Deliverable demo . . . 14
3.2.1 Demo structure . . . 14
3.2.2 Text Pre/Post-processing . . . 14
3.2.3 Translation servers . . . 15
3.2.4 Load Balancer . . . 15
3.2.5 How To Run . . . 16
3.3 Improved BLEU scores . . . 17
3.4 Integration with neural LM . . . 17
3.5 Tuning scripts . . . 17
3.5.1 Tuning test sets . . . 18
3.5.2 Run parameter tuning . . . 18
3.5.3 Check on tuning progress . . . 20
3.5.4 Stop tuning . . . 23
3.6 Text processing scripts . . . 24
3.7 Empirical evaluation . . . 25
3.7.1 Systems . . . 26
3.7.2 Setup . . . 28
3.7.3 Results . . . 31

4 Conclusions **42**

Appendices **45**

A Test machine configurations (smt10) **45**

1 Introduction

This REMEDI deliverable extends the previous deliverable, described in [ISZ16], by:

- **Communication protocols** - the communication protocols description for 3-rd party integration;
- **Delivery Demo** - config files, models, scripts, and instructions for the complete infrastructure demo;
- **Improved BLEU scores** [PRWZ02] - more than 30% improvement in BLEU due to bug fixes;
- **Integration with neural LM** - a test integration with Neural Language model;
- **Tuning scripts** - integrated scripts for parameter tuning;
- **Text processing scripts** - example scripts using NLTK [LB02] and Stanford Core NLP [MSB+14];
- **Empirical evaluation** - multi-threading performance of Oister, REMEDI and Moses [KHB+07];

As before, our software keeps following the client/server architecture based on the WebSockets protocol [Web11]. For this deliverable, our software package was completed with the following top-level scripts, turning it into a complete translation system infrastructure:

- **run_tuning.sh** - the script for starting the tuning process;
- **tuning_progress.pl** - the script for monitoring the tuning progress and retrieving the best scoring configuration files;
- **kill_tuning.pl** - the script for killing the tuning infrastructure scripts, is used to stop the tuning process;
- **pre_process_nltk.sh** - an example script allowing for language detection and text preprocessing;
- **post_process_nltk.sh** - an example script allowing for text post-processing;
- **start.sh** - the infrastructure demo running script;

The rest of the report is organized as follows: Section 2 describes the structure of the deliverable. Section 3 provides all necessary details about the current delivery including data on the new software pieces and their usage. Section 4 concludes and indicates possible future work directions.

2 Deliverables

This deliverable consists of this report as well as software deliverables. Both are distributed as a downloadable archive with the following standard structure that will be re-used for the subsequent deliveries:

- **REMEDI/month-24/data/** - stores the delivery's demo
- **REMEDI/month-24/software/** - stores the software project
- **REMEDI/month-24/report/** - stores this document

Next, we provide some details on the structure of the software part of the delivery. The software components are located in `REMEDI/month-24/software/`, and are accompanied by an extended markdown [Gru15] document file `README.md` providing all the necessary software details. Similar, but shorter, information is also provided in Section 3 of this document. The delivered software is a standard Netbeans 8.0.2 C++ project, that can also be build using cmake and make, and its top-level structure is as follows:
[Project-Folder]/

doc/ - project-related documentation

ext/ - external libraries used in the project

inc/ - C++ header files

src/ - C++ source files

script/ - stores various scripts

script/web/ - web client for translation system

script/text/ - example pre/post-processing scripts

script/tuning/ - scripts and software required for tuning

script/test/ - scripts used for testing

nbproject/ - Netbeans project data

`server.cfg` - example server configuration file

`balancer.cfg` - example load balancer configuration file

`processor.cfg` - example processor configuration file

`LICENSE` - code license (GPL 2.0)

`CMakeLists.txt` - the cmake build script

`README.md` - software information document

`Doxyfile` - Doxygen configuration file

3 Delivery Details

In this section, we discuss the main aspects of the delivered software. This is a cumulative release, so for the information on licensing, software structure, building, and running[1] we refer to the previous delivery report [ISZ16] and the software documentation [Zap17]. In this section we will concentrate on the new functionality and provide a few details of the implementation aspects. As before, we produce an open-source and freely-distributed GPL2.0 (http://www.gnu.org/licenses/) licensed software that is available as a github project:

https://github.com/ivan-zapreev/Basic-Translation-Infrastructure

The rest of the section is organized as follows: Section 3.1 describes the "JSON over WebSockets" - based communication protocol used in the infrastructure. Section 3.2 gives a short introduction of the demonstration scripts delivered with the project. In Section 3.3 we mention the BLEU score improvements obtained by fixing software bugs. Neural Language Model integration experiments are briefly described in Section 3.4. Further, Section 3.5 introduces the tuning scripts added to this deliverable. The example text processing scripts created for the project are discussed and explained in Section 3.6. Finally in Section 3.7 we provide a detailed experimental comparison of performance between REMEDI, Oister and Moses translation systems.

3.1 Communication protocols

In this section we describe the communication protocols between the system applications, see also Sections "General design" and "Communication protocols" of [Zap17]. As it has already been noted, all communications between the `bpbd-client`, `bpbd-server`, `bpbd-balancer`, `bpbd-processor`, and `translate.html` are based on the WebSockets [Web11] communication protocol. The latter is a rather low-level messaging protocol allowing for asynchronous interaction between applications. In our framework it is used as a data transfer protocol we build our communications on. We chose the communicated message's payload to be formed using JavaScript Object Notation (JSON) [JAS06] which is a lightweight human-readable data-interchange format.

In essence, our applications interact by sending JSON message data objects to each other over WebSockets. There are just three types of communications (request-response messages) present in the system at the moment:

- **Supported languages - (SL)**: is used when one application needs to retrieve supported translation source/target language pairs from another application;
- **Translation - (T)**: is used when one application requires another application to perform some text translation from source language into a target one;
- **Pre-/Post-processing - (PP)**: is used when one application needs to pre/post-process some text, before/after translating it, in the given source/target language;

[1] Running of the previously delivered software components.

Request Response	bpbd-server	bpbd-balancer	bpbd-processor
bpbd-client	(SL), (T)	(SL), (T)	(PP)
translate.html	(SL), (T)	(SL), (T)	(PP)
bpbd-balancer	(SL), (T)	(SL), (T)	

Table 1: Application communication types

In the list above, we did not mention concrete application names as some of the communication types are common for multiple applications. To get a better insight into which communications are possible, consider Table 1. The left-most column contains applications that can initiate this or that communication request. The top-most row contains applications that can respond to this or that request with a proper response. The non-present relations or empty cells in the table indicate no possible communications.

As shown in the table above, bpbd-client and translate.html can only initiate requests, whereas bpbd-server and bpbd-processor can only produce responses. At the same time, bpbd-balancer does not only produce responses for bpbd-client and translate.html, but can also initiate requests for other instances of bpbd-balancer or bpbd-server applications.

Below we consider each of the aforementioned communication types in more details. Consider Section 3.1.2 for (SL), Section 3.1.3 for (T), and Section 3.1.4 for (PP) communications. In these sections we will also provide the protocol's JSON formats to facilitate development of/communication to the third-party client and/or server applications. Let us start Section 3.1.1 which describes the common base class used for all communication messages.

3.1.1 Common base class

In the actual C++ code design given in Figure 1, one can see that we have used intricate multiple inheritance for the request and response message classes.

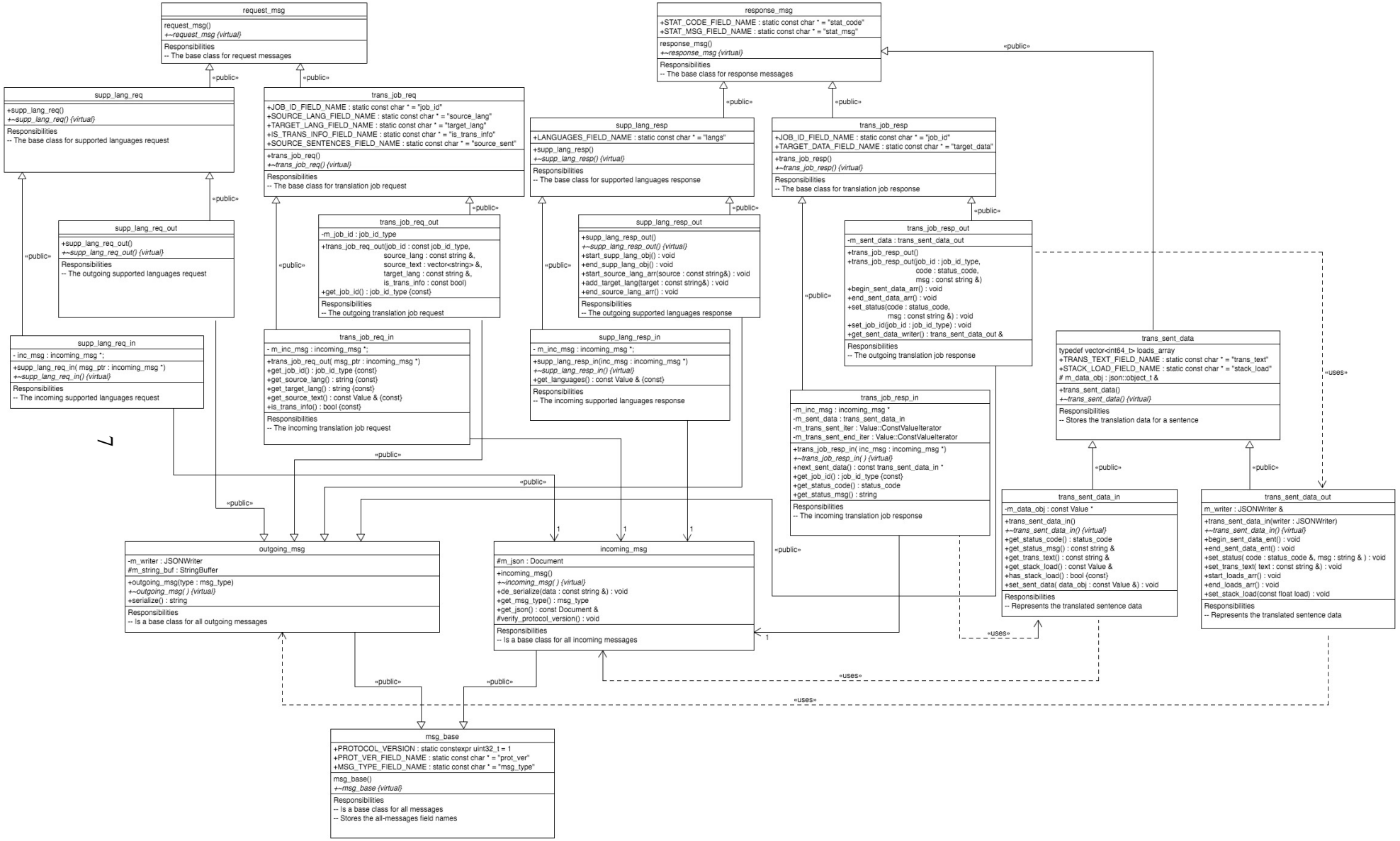


Figure 1: Messaging classes

Yet, when serialized into the JSON format the situation is much simpler. Consider the JSON object given below. It can be seen as an instance of the common base class of all the communication messages we use in our software:

```
{
  "prot_ver" : 0,
  "msg_type" : 5
}
```

Clearly, this base class has just two compulsory fields which are:

- `prot_ver` - an *unsigned integer* indicating the protocol version;
- `msg_type` - an *integer* indicating the message type;

The currently available message types are specified by the following C++ enumeration values:

```
enum msg_type {
  //The message type is undefined
  MESSAGE_UNDEFINED = 0,
  //The supported languages request message
  MESSAGE_SUPP_LANG_REQ = 1,
  //The supported languages response message
  MESSAGE_SUPP_LANG_RESP = 2,
  //The translation job request message
  MESSAGE_TRANS_JOB_REQ = 3,
  //The translation job response message
  MESSAGE_TRANS_JOB_RESP = 4,
  //The pre-processor job request message
  MESSAGE_PRE_PROC_JOB_REQ = 5,
  //The pre-processor job response message
  MESSAGE_PRE_PROC_JOB_RESP = 6,
  //The post-processor job request message
  MESSAGE_POST_PROC_JOB_REQ = 7,
  //The post-processor job response message
  MESSAGE_POST_PROC_JOB_RESP = 8
};
```

All of the message classes discussed below inherit the `prot_ver` and `msg_type` fields.

3.1.2 (SL) - Supported languages

The supported-languages request is used to retrieve the list of the supported source and target language pairs. At present, each `bpbd-server` only supports one source/target language pair. Being an aggregator of multiple translation services, each `bpbd-balancer` can support multiple language pairs.

JSON Request format: The request for supported languages does not extend the common base class, as explained in Section 3.1.1, with any new fields. All it takes to create an (SL) request is to instantiate the base class object with the proper value of its fields. See the example below which has the `msg_type` set to 1, being the message type of (SL) requests.

```
{
  "prot_ver" : 0,
  "msg_type" : 1
}
```

JSON Response format: The (SL) response is supposed to return the list of supported language pairs. Clearly one source language can be translated into multiple target languages. This is illustrated with the following (SL) response example object.

```
{
  "prot_ver" : 0,
  "msg_type" : 2,
  "langs" :
   {
      "german" : [ "french", "english"],
      "chinese" : [ "english" ]
   }
}
```

In this example, we see that `msg_type` is set to 2, indicating the this is an (SL) response and also there is an additional field `langs` which is an *object* storing the source to target language mappings. There, the object's field name indicates the source language and its value - which has to be an *array of strings* - stores the list of target languages. If the source language can not be translated in any language then it should not be present. In other words, an empty array of target languages is not supported.

Note that, the (SL) response does not support returning an error. This might be introduced in the latter protocol versions, but for now it is expected that each (SL) request should be handled properly. There is just two possibilities when it can fail:

1. The service does not support this type of request;
2. The service does support this type of requests but it is down;

Both of the above cases are currently considered as exceptional and only occur during development or setting up the infrastructure. Thus they are handled by simply returning a text error message through the WebSocket.

3.1.3 (T) - Translation

Translation requests and responses are used to communicate the source text to the translation service and the resulting target text to its client. Clearly the source and target texts can be large

and therefore our protocol supports splitting those texts into multiple translation requests and responses correspondingly.

JSON Request format Each source text to be translated is split into a number of translation jobs, consisting of a number of translation tasks - being sentences. Within the client and server applications it is important to be able to keep track of all individual translation job requests, per client, to be able to identify which of them has been replied or not. There are also other things that need to be in the (T) request, such as: the request priority, the source and target language pair and etc. To see them all let us consider the (T) request example object below:

```
{
  "prot_ver" : 0,
  "msg_type" : 3,
  "job_id" : 101,
  "priority" : 10,
  "source_lang" : "german",
  "target_lang" : "english",
  "is_trans_info" : true,
  "source_sent" : [
    "how are you ?",
    "i was glad to see you .",
    "let us meet again !"
  ]
}
```

In the example above we have:

- `job_id` - an *unsigned integer* indicating the translation job id unique for the given client;
- `priority` - an *integer* storing the job priority where 0 means neutral and higher values mean higher priority;
- `source_lang` - a *string* with the source language;
- `target_lang` - a *string* with the target language;
- `is_trans_info` - a *boolean* flag indicating whether we need to get an additional translation information with the job response. Such information includes but is not limited by the multi-stack loads and gets dumped into the translation log;
- `source_sent` - an *array of strings* which are sentences to be translated, appearing in the same order as they are present in the original text;

Note that, there is no limit on the number of sentences to be sent per request. However, the provided client implementations split the original text into a number of requests to improve the system's throughput in the multi-client environment.

JSON Response format The translation job response message is supposed to indicate which translation request it corresponds to, contain the overall request status and the translated text, consisting of target sentences. The latter are attributed with translation status and (optionally) translation info, giving some insight into the translation process. Let us consider an example translation response below.

```
{
  "prot_ver" : 0,
  "msg_type" : 4,
  "job_id" : 101,
  "stat_code" : 3,
  "stat_msg" : "Some translation tasks were canceled",
  "target_data" : [
     {
       "stat_code" : 2,
       "stat_msg" : "OK",
       "trans_text" : "Wie geht es dir ?",
       "stack_load" : [ 3, 67, 90, 78, 40, 1 ]
     },
     {
       "stat_code" : 4,
       "stat_msg" : "The service is going down",
       "trans_text" : "i was glad to see you ."
     },
     {
       "stat_code" : 2,
       "stat_msg" : "OK",
       "trans_text" :  "Lass uns uns wiedersehen!",
       "stack_load" : [ 7, 76, 98, 90, 56, 47, 3 ]
     }
   ]
}
```

In the example above, the status code is defined by the following C++ enumeration values:

```
enum stat_code {
  //The status is not defined
  RESULT_UNDEFINED = 0,
  //The status is unknown
  RESULT_UNKNOWN = 1,
  //The translation was fully done
  RESULT_OK = 2,
  //Some sentences in the translation job were not translated
  RESULT_PARTIAL = 3,
  //The entire translation job/task was canceled
```

```
  RESULT_CANCELED = 4,
  //We failed to translate this entire translation job/task
  RESULT_ERROR = 5
};
```

Note that `stat_code` and `stat_msg`, storing the translation status, are given at the top level of a translation job—indicating the overall status—and also at the level of each sentence. Also, `stack_load`, storing an array of stack loads in percent, is only present for a translated sentence if a translation info was requested. The latter is done by setting the `is_trans_info` flag in the corresponding translation job request. The order of translated sentence objects in the `target_data` array will be the same as the order of the corresponding source sentences in the `source_sent` array of the translation job request.

3.1.4 (PP) - Pre/Post processing

Text processing requests and responses are used to communicate the source/target texts to the text processing service for pre and post processing. Clearly the source and target texts can be large and therefore our protocol supports splitting those texts into multiple (PP) requests and responses. In case of text processing, we cannot split a text in an arbitrary language into sentences at the client side. This would be too computationally intensive and would also require presence of corresponding language models at the client. Therefore, it has been decided to split text into UTF-8 character chunks of some fixed length. Let us consider the (PP) requests and responses in more details.

JSON Request format: An example (PP) request is given below. Here we give the pre-processor job request as indicated by the value of `msg_type`.

```
{
  "prot_ver" : 0,
  "msg_type" : 5,
  "job_token" : "176393e9aae9b108767fafda109f4620",
  "priority" : 0,
  "num_chs" : 5,
  "ch_idx" : 1,
  "lang" : "auto",
  "text" : "Als die Schule aus Gr\UTF{00FC}nden lernen.
            Haben Sie Freunde kommen aus der
            Ferne? Ist es nicht \UTF{00E4}rgerlich, nicht
            haben."
}
```

In the example object above we have the following fields:

- `job_token` - a *string* representing a semi-unique token specific for the text to be processed for the given client. For the pre-processor request (`msg_type == 5`) the initial value of `job_token` is chosen to be the MD5 sum of the entire text to be pre-processed.

The pre-processor server can update the token to make it client-session specific. The (updated) token is always returned with the corresponding processor response. This new value is to be used for any subsequent post-processing request of the target text corresponding to the given source text.

- `priority` - a *integer* value of the job priority, the same meaning as for the (T) requests;
- `num_chs` - a *unsigned integer* giving the number of chunks the text is split into;
- `ch_idx` - a *unsigned integer* indicating the index of the current chunk, starts from zero;
- `lang` - a *string* storing the language of the text, or "auto" for the language auto detection. The latter is only allowed in case of the pre-processor job request, i.e. when `msg_type == 5`;
- `text` - a *string* storing text characters of the given chunk;

JSON Response format: An example (PP) response is given below. Here we give the pre-processor job response as indicated by the value of `msg_type`.

```
{
  "prot_ver" : 0,
  "msg_type" : 6,
  "stat_code" : 2,
  "stat_msg" : "Fully pre-processed",
  "job_token" : "176393e9aae9b108767fafda109f4620.1456",
  "num_chs" : 7,
  "ch_idx" : 1,
  "lang" : "chinese",
  "text" : "als die schule aus gr\UTF{00FC}nden lernen .
            haben sie freunde kommen aus der ferne ?
            ist es nicht \UTF{00E4}rgerlich , nicht haben ."
}
```

In the example object above we have the following fields:

- `stat_code` - a *unsigned integer* indicating the status code of the response, the same meaning as for (T) responses;
- `stat_msg` - a *string* storing the response status message, the same meaning as for (T) responses;
- `job_token` - a *string* containing the job token. For the pre-processor response, it is updated by the text processing service: The original job token sent with the pre-processor request is made unique to the user session. This updated tokens is then to be used for any subsequent post-processing requests of the target text corresponding to this source text. For the post-processor response, the value of the job token does not change and is the same as sent with the post-processor request;

- `num_chs` - a *unsigned integer* indicating the number of chunks in which the pre/post-processed text is split for being sent back to the client;
- `ch_idx` - a *unsigned integer* value of the current chunk index, starts from zero;
- `lang` - a *string* storing the text's language. In case the pre-processor request was sent with the `lang` field value set to "auto", the pre-processor service supports language detection, and the language was successfully detected, this field will contains the name of the detected source language;
- `text` - a *string* storing text characters of the given chunk;

3.2 Deliverable demo

The purpose of the delivered demo is to show the functionality of the entire translation infrastructure. The demo is meant to be used with the web-based translation client. All application instances are run on the same machine.

Note that, all of the server applications run in this demo will be run using the Linux `screen` command. I.e. after running the demo's `start.sh` script one can use `screen -r` to see all of the running screens and then to connect to any of them using the `screen -r <screen-id>` command. Once connected you can see which server is running inside this screen, observe its status, and interact with its console.

The rest of the section is organized as follows. Section 3.2.1 provides a brief description of the demo folder's content. Next Sections 3.2.2, 3.2.3, and 3.2.4 give information on the configuration of the text processor, translation, and load balancing servers. Finally, Section 3.2.5 lists the steps needed to run the demo.

3.2.1 Demo structure

Consider a brief description of the content of the demo's folder:
[Demo-Folder]/

`configs/` - server configuration files

`models/` - pre-trained model files

`test/` - source/target texts

`ext/` - additional information folder

`start.sh` - demo starting script

`READ.ME` - demo's information document

Additional information on the demo's directory structure can be found in the `READ.ME` file.

3.2.2 Text Pre/Post-processing

The text processing server, run on port 9005, is configured to support the following features:

(A) Language detection;

(B) Sentence splitting;

(C) Tokenization;

(D) Lower-casing;

(E) De-tokenization;

(F) Upper-casing;

(G) Structure restoration;

The supported source languages for are: Chinese, Dutch, Finnish, Greek, Polish, Spanish, Czech, English, French, Italian, Portuguese, Swedish, Danish, Estonian, German, Norwegian, Slovene, Turkish. The supported target languages are: English. Please note that, the features described above are implemented within this projects' delivery and can be used in production, see Section 3.6 for more details.

3.2.3 Translation servers

The translation, reordering, and language models used in this demo's decoding servers are very small and thus have very limited translation power. The only remotely powerful set of models is used for the Chinese to English server. This set of models is capable of translating the Chinese text provided with the demo. The models for it were obtained by filtering the original fill-power Chinese to English models to just be capable of translating the given source text. The models used for German to English translations are just too small and are used mostly to have some sort of dummy German to English server running. The models used for German to French translation servers are the same as used for German to English. So no real German to French translation is possible. Note that, we run four translation/decoding servers in total:

- One German to French server running on port 9001;
- One Chinese to English server running on port 9002;
- Two German to English servers running on ports 9003 and 9004;

3.2.4 Load Balancer

All of the decoding servers in this demo, although accessible individually, are put behind a load balancer for two reasons:

1. To show how workload can be distributed between two German to English servers
2. To show the Web-client functionality allowing to choose between multiple source and target languages.

The load balancing server is configured to run on port 9007.

3.2.5 How To Run

Let us consider the main steps needed to run this demo:

1. Modify the `REMEDI_PATH` variable of the `start.sh` script to point to the REMEDI project folder.
2. Download the Stanford Core NLP distribution (we used version 3.7.0) from http://stanfordnlp.github.io/CoreNLP/
3. Extract the Stanford Core NLP from the downloaded archive.
4. Download the corresponding version's Chinese language module jar, from the same http://stanfordnlp.github.io/CoreNLP/, and place it inside the previously extracted Stanford Core NLP folder.
5. Modify the `CORE_NLP_PATH` variable of the `start.sh` script to point to the the previously downloaded Stanford Core NLP folder.
6. Execute the `start.sh` script. It will start the required application servers. Please note that, even though the translation models are not that big, it can take several minutes until they are loaded and the servers' infrastructure is ready for accepting translation requests.
7. Open `$REMEDI_PATH/script/web/translate.html` (web-client) in a web browser.
8. Make sure that the "Connections" section of the web-client page is set up as:
   - Pre-processor: ws://localhost:9005
   - Translator: ws://localhost:9007
   - Post-processor: ws://localhost:9005
9. Check that the"Source" drop-down box of the web-client contains both Chinese and German languages. The latter is the sign that the models are loaded, the servers are running and ready to accept translation requests. In case the source languages are not present, try reloading the web-client page until they will appear. Note that re-loading the web-client page can re-set the server connection parameters in the "Connections" section. In case the source languages do not appear, please use the `'screen -r'` command to connect to the screens running the servers to check on their status.
10. In the web-client use the Browse button to load in the Chinese text from:

    ```
    ./test/chinese.head.10.txt
    ```

    Keep the source language selection to be set to: `--detect--`. Choose the target language to be English and press the translate button.

11. Wait until the translation is finished and compare the resulting translation with the one stored in `./test/english.head.10.full.lm.txt`. You will see that the translations from the file are somewhat better. The difference comes from the fact that this file was generated using a larger English language model.

12. Note that, the Chinese language was detected automatically - the source language is now set to Chinese. The structure of the resulting English translation follows the structure of the original Chinese text. Also, the words capitalization in the target text is reasonable.

13. Press `'Enter'` in the console running the `start.sh` script. This will terminate the running infrastructure.

3.3 Improved BLEU scores

In our experiments done for the previous deliverable [ISZ16] we have reported that for the Chinese to English translation on the **MT-**04 dataset [Gro10], with the CTB segmented [MSB+14] text consisting of 1788 sentences and 49582 tokens, REMEDI was not scoring as high as Oister. In fact, tuned on the same data and the values of the non-tuned parameters matched, c.f. Section 3.7.2, REMEDI was getting just $\approx$ 23 BLEU points whereas Oister achieved 36.8. After fixing a few issues in the decoder's code, under the same settings as before, REMEDI is now showing the BLEU score of 36.72. This new value indicates the translation quality provided by REMEDI match that of Oister.

3.4 Integration with neural LM

An experimental neural-network based language model integration was done by a fellow PhD student from the ILPS group of the University of Amsterdam: Hamidreza Ghader, with the direct assistance of the REMEDI project members. Since the advantage of the developed language model is not confirmed yet, it is not included in the REMEDI release. During the integration of the language model the internal interfaces were followed in the same way as it was done when implementing generic tries, see Section 3.5.2 of [ISZ15]. The only modification needed, to make the neural language model work, from the side of the core REMEDI code, was: Allowing the language model query class to store the probability and back-off data within itself, rather than storing a reference to the data. The reason for this change was that each trie implementation created previously would explicitly store the n-gram probabilities and back-off weights within the trie data structure. Thus it was enough to only store the reference to this data in the query class. The neural model does not store the explicit probability values per n-gram but rather computes them each time for each queried n-gram. Therefore, explicit data storing rather than reference storing became needed. The language model query class was modified accordingly to accommodate this new requirement. The change was tested and is proven to have no observable impact on the overall decoding times.

3.5 Tuning scripts

In order to obtain the best performance of the translation system one can employ Discriminative Training, see Chapter 9 of [Koe10]. The latter uses generated word lattices, c.f. Chapter 9.1.2

of [Koe10], to optimize translation performance by reducing some measure of translation error. This is done by tuning the translation parameters such as feature lambda values of the model feature weights.

We measure the translation system performance in terms of the BLEU scores. Therefore, parameter tuning will find such lambda values, to be used in the `bpbd-server` configuration files that they maximize the BLEU scores of the translations provided by the system. As was already mentioned in our previous report [ISZ16], our software allows for word lattice generation.

In this section we explain the scripts we use to perform parameter tuning. Please note that, the tuning scripts are implemented using Perl and bash. We expect the Perl version to be $\geq v5.10.1$. Also, in order to use the tuning scripts one must have MegaM compiled, see Section "Building the project" of [Zap17] for the corresponding build instructions.

The rest of the section is organized as follows. First in Section 3.5.1, we report on the test-sets that we use for tuning and their internal structure. Next in Section 3.5.2, we explain how the tuning script can be run. Section 3.5.3 reports on how the tuning progress can be monitored and the server configuration files can be generated for various tuning iterations. At last in Section 3.5.4, we explain how tuning can be aborted in an easy manner.

3.5.1 Tuning test sets

Parameter tuning is often done using NIST Open Machine Translation (OpenMT) Evaluation test sets. Typically, we use the MT-04 dataset [Gro10]. From these test sets we use two types of files:

- **Source file** - a pre-processed text in the source language, ready to be translated;
- **Reference translation files** - reference translation files, in the target language, not post-processed;

The source files are to be plain text files, pre-processed in the same way as when running the translation infrastructure. I.e., using the same sentence splitting methods and tokenizers. Of course, the pre-processed text is expected to be lower-cased and to have one sentence per line. See Section 3.6 for more details on text processing.

The reference files also have to be in plain text format. Moreover, they are expected to be lower-cased, tokenized and have one sentence per line. I.e., to have the same format as the text produced by the `bpbd-server`. We support multiple reference translations and therefore all the translation files must have the same file name format: `<file_name_base>.<ref_index>`. Here `<file_name_base>` is the same file name used for all reference translation files of the given source text. The `<ref_index>` is the reference translation index, starting from 0. Note that, even if there is just one reference translation then it will still get the index in its file name. The maximum number of reference translation files is 100.

3.5.2 Run parameter tuning

In order to start parameter tuning one should use the `run_tuning.sh` script located in

```
[Project-Folder]/scripts/tuning/
```

If run without parameters, it gives the following usage information:

```
$ ./run_tuning.sh
------
USAGE: Allows to start the tuning process for the
              infrastructure
------
 ./run_tuning.sh --conf=<file-name> --src=<file-name>
        --src-language=<string>  --ref=<file-pref>
        --trg-language=<string> --no-parallel=<number>
        --trace=<number> --nbest-size=<number>
        --mert-script=<string>
 Where:
    --conf=<file-name> the initial configuration file for the
                           decoding server
    --src=<file-name> the source text file to use with tuning
    --src-language=<string> the language of the source text
    --ref=<file-pref> the reference translation text file name
                           prefix. The files names are
                           constructed as:
                                  <file-templ>.<index>
                           where <index> starts with 1.
    --trg-language=<string> the language of the reference text,
                          default is english
    --no-parallel=<number> the number of parallel threads used
                          for tuning, default is 1
    --trace=<number> the tracing level for the script, default
                          is 1
    --nbest-size=<number> the number of best hypothesis to
                          consider, default is 1500
    --mert-script=<string> the MERT script type to be used,
                          default is 'PRO-14'
```

As one can see the script's interface is fairly simple. It requires the name of the config file, the source text file, the source language name, the target language name, and the reference file(s) prefix. Other parameters are optional and can be omitted. Note that, when run on a multi-core system, for the sake of faster tuning, it is advised to set the value of the `--no-parallel=` parameter to the number of cores present in the system. More details on the source and reference files can be found in Section 3.5.1.

Please note that for the tuning script to run, we need the initial `bpbd-server` configuration file. The latter can be obtained by copying the default `server.cfg` file from the project's folder to the work folder, where tuning will be run, and then modify it with the proper source and

target language names, the model file locations, and modify other values, if required. The tuning script itself can be run from any folder, chosen to be a work folder. Let us consider an example invocation of the tuning script:

```
$ [Project-Folder]/script/tuning/run_tuning.sh --conf=server.cfg \
   --no-parallel=39 --trace=1 --nbest-size=1500 \
   --src-language=chinese --src=$source.txt \
   --ref=$reference.txt --trg-language=english
-----
Use '[Project-Folder]/script/tuning/tuning_progress.pl'
to monitor the progress and retrieve optimum config.
Use '[Project-Folder]/script/tuning/kill_tuning.pl'
to stop tuning.
-----
Starting tuning on: smt10.science.uva.nl at:
Fri Dec 16 17:07:00 CET 2016
Writing the tuning log into file: tuning.log ...
```

Here, we explicitly specified the optional parameters: `--no-parallel=39`, `--trace=1`, and `--nbest-size=1500`. Once the script is started, the tuning process output is written into the `tuning.log` file. The output of the used `bpbd-client` is placed in the `client.log` and the output of the lattice combination script into the `combine.log`. Note that, the tuning script blocks the console and runs until one of the conditions is satisfied:

- The maximum allowed number of tuning iterations is done, this value is set to 30;
- The tuning process is stopped by the `kill_tuning.pl` script, see Section 3.5.4;

3.5.3 Check on tuning progress

Tuning can takes several hours to a couple of days, depending on the number of features, the size of the models, the size of the development set, etc. Therefore, we have created the `tuning_progress.pl` script, located in `[Project-Folder]/script/tuning/`, that allows to monitor the tuning progress. When run without parameters this script provides the following output:

```
$ ./tuning_progress.pl
Smartmatch is experimental at ./tuning_progress.pl line 100.
-------
INFO:
    This script allows to monitor running/finished tuning
    process and to extract the configuration files for
    different tuning iterations. This script must be run
    from the same folder where tuning was/is run.
```

```
-------
USAGE:
    tuning_progress.pl --conf=<file_name> --err=<file_name>
    --select=<string>
    Where:
        --conf=<file_name> - the configuration file used in
                        tuning process
        --err=<file_name>  - the tuning.log file produced
                        by the tuning script
        --select=<string>  - the iteration index for which the
                         config file is to be generated or
                         'best'/'last' values to get the config
                         files for the best-scoring or last
                         iteration respectively. This
                         parameter is optional, if specified
                         - the script generates a corresponding
                         iteration's config file
-------
ERROR: Provide required script arguments!
```

As on can notice this script can be used for two purposes:

1. Monitor the tuning progress;
2. Generate the configuration of a certain tuning iteration;

Let us consider both of these usages in more details.

Monitor Progress Invoke the `tuning_progress.pl` script from the work folder where the tuning is run. Use the `tuning.log` log file as the value of the `--err=` parameter and the used config file as the value of the `--conf=` parameter:

```
$ [Project-Folder]/script/tuning/tuning_progress.pl \
      --conf=server.cfg --err=tuning.log
Avg. total = 0.00 sec.

Avg. total = 0.00 sec.

iteration=1     BLEU=0.297540050683629
iteration=2     BLEU=0.304940039112847  ^
iteration=3     BLEU=0.31246539393292   ^
iteration=4     BLEU=0.319738621056022  ^
iteration=5     BLEU=0.324859305442724  ^
iteration=6     BLEU=0.331238665525784  ^
```

```
iteration=7     BLEU=0.337603780346243  ^
iteration=8     BLEU=0.341755568020084  ^
iteration=9     BLEU=0.343932768784104  ^
iteration=10    BLEU=0.350114574042927  ^
iteration=11    BLEU=0.351529162662153  ^
iteration=12    BLEU=0.354527741244185  ^
iteration=13    BLEU=0.358353976275115  ^
iteration=14    BLEU=0.359557421035731  ^
iteration=15    BLEU=0.362479837703255  <--- best overall BLEU
iteration=16    BLEU=0.362085630174003  v
iteration=17    BLEU=0.36209376384702   ^
iteration=18    BLEU=0.359820528770433  v
iteration=19    BLEU=0.356861833879925  v
iteration=20    BLEU=0.354888166740936  v
iteration=21    BLEU=0.353153254195068  v
iteration=22    BLEU=0.355045301787933  ^
iteration=23    BLEU=0.351972501936527  v
iteration=24    BLEU=0.348495504435742  v

(1) de_lin_dist_penalty
(2) lm_feature_weights
(3) rm_feature_weights
(4) tm_feature_weights
(5) tm_word_penalty

...
```

This will print out a lot of information, but the most relevant part is shown above. It indicates the tuning iterations with the corresponding BLEU scores. Here tuning was running for a while already and we can see that for iterations 1-15 BLEU scores went up (the ^-sign), with iteration 15 yielding the best BLEU score. After that (iterations 16-24), BLEU scores went (mostly) down.

As a rule of thumb, if you see that BLEU scores drop for two consecutive iterations after the optimal iteration, it is time to stop tuning. The latter can be done with the `kill_tuning.pl` script discussed in Section 3.5.4;

Generate config When the `run_tuning.sh` script is still running, or after its execution is finished, one can generate the config file corresponding to any of the performed tuning iterations, as listed by the `tuning_progress.pl` script. In order to do so, in addition to the `--conf=` and `--err=` parameters one can just specify the third one: `--select=`, supplying the iteration number. For example, to generate the configuration script used in the best scoring tuning iteration of the example above, one can use the following script invocation:

```
$ [Project-Folder]/script/tuning/tuning_progress.pl \
   --conf=server.cfg --err=tuning.log --select=best
```

or alternatively:

```
$ [Project-Folder]/script/tuning/tuning_progress.pl \
   --conf=server.cfg --err=tuning.log --select=15
```

Since iteration 15 was the best scoring one, these will result in two identical configuration files being generated: `server.cfg.best` and `server.cfg.15`. Please note that, the `tuning_progress.pl` script must be run from the same folder as the `run_tuning.sh` script as it uses some hidden temporary files stored in the work folder.

3.5.4 Stop tuning

If the best scoring tuning iteration has been reached one might want to stop tuning right away. This could be done by killing the `run_tuning.sh` script but things are a bit more complicated than that. The tuning script forks plenty of independent processes each of which log its PID value into the `tuning.log` file. Clearly, if `run_tuning.sh` is killed this will not affect the forked processes. This is why we have developed the `kill_tuning.pl` script located in the `[Project-Folder]/script/tuning/` folder. The only parameter required by the script is the `tuning.log` file used as a source of the related PID values. The script can be invoked as follows:

```
$ [Project-Folder]/script/tuning/kill_tuning.pl tuning.log

Are you sure you want to stop the tuning process?

Answer (yes/no)? yes
Starting killing the active tuning processes ...
-----------------------------------------------------
Starting a killing iteration, analyzing the error log
for process ids. The error log analysis is done,
starting the killing.
 8380 pts/2    00:00:00 tuner.pl
 8385 pts/2    02:01:30 PRO-optimizatio
 4823 pts/2    01:57:44 bpbd-server
15260 pts/2    00:00:00 PRO-optimizer-p
15265 pts/2    00:05:37 PRO-optimizer-l
...
15297 pts/2    00:05:38 PRO-optimizer-l
15303 pts/2    00:05:38 PRO-optimizer-l
The number of (newly) killed processes is: 18
-----------------------------------------------------
```

```
Starting a killing iteration, analyzing the error log
for process ids. The error log analysis is done,
starting the killing.
The number of (newly) killed processes is: 0
-----------------------------------------------------
The killing is over, we are finished!
```

Please note that, the script requires you to type in `yes` as a complete word and press enter to start the killing. Any other value will cause the killing to be canceled.

3.6 Text processing scripts

As was described in Section 3.2.5 of [ISZ16], our translation framework contains a text processing server that can be used for text pre/post-processing for before/after translation. This server can be configured to use virtually any third-party text processing software. In the previous deliverable we only supplied dummy pre/post-processing scripts `pre_process.sh` and `post_process.sh` that did not much more than copying input text into output.

To make our software complete and also to show how third-party pre/post-processing software can be integrated into our project we have created new example pre/post-processing scripts, both of which are Natural Language Toolkit (NLTK) [LB02] based, and thus require python and NLTK for python to be installed. The installation instructions are simple and are to be found on the toolkit's website. Let us consider the created scripts:

- `./script/text/pre_process_nltk.sh`:
  - NLTK stop-words analysis [COJA15] to detect languages.
  - Text template generation for text structure restoration.
  - Text tokenization and lowercasing as provided by NLTK.
  - Sentence splitting using NLTK and Stanford Core NLP [MSB+14]
- `./script/text/post_process_nltk.sh`:
  - Sentence capitalization as a separate option.
  - Text de-tokenization based on MTMonkey [AORP13] de-tokenization scripts.
  - Text true-casing based on Moses [KHB+07] or Truecaser (https://github.com/nreimers/truecaser) scripts based on [LIRK03].
  - Text structure restoration from a pre-generated template.

These scripts call on python or Perl scripts delivered with the distribution. The latter are configurable by their command-line parameters which are typically well-documented. In order to change the default scripts' behavior we expect our users to edit these parameters inside the `pre_process_nltk.sh` and `post_process_nltk.sh` scripts. It is also important to note that:

- The text structure restoration is per default enabled but it is then expected that both pre- and post-processing scripts are run on the same file system with the same work directory.
- The text structure restoration can be disabled by skipping the `[-t TEMPL]` command-line parameter for the `pre_process_nltk.py` and `post_process_nltk.py` scripts.
- The interface of `pre_process_nltk.sh` is the same as that of `pre_process.sh`.
- The list of `post_process_nltk.sh` parameters exceeds that of `post_process.sh`. The former requires the `<true_caser_type>` parameter to be specified and also has an optional parameter `<models-dir>`. Run `post_process_nltk.sh` with no parameters to get more info.
- The `<true_caser_type>` parameter of `post_process_nltk.sh` allows to enable one of the two true-caser scripts: Moses or Truecaser.
- The `<models-dir>` parameter of `post_process_nltk.sh` is optional but defines the folder where the true-caser model files are to be found, the default is "`.`", i.e., the current folder.
- The true-casing models are supposed to have file names as the lower-cased English names of the corresponding languages. The model file extensions are supposed to be `*.tcm` for Moses and `*.obj` for Truecaser, e.g.: `english.tcm` or `chinese.obj`.
- Our project does not provide any default true-caser models neither for Moses nor for Truecaser. So for `post_process_nltk.sh` to be used with `<true_caser_type>` something other than 'none' one needs to obtain such model(s) for used target language(s).
- In order to generate new true-caser models, one can use the corresponding training software scripts provided with the distribution:
  - Moses: `./script/text/truecase/moses/train-truecaser.perl`
  - Truecaser: `./script/text/truecase/truecaser/TrainTruecaser.py`

  These scripts are taken 'as-is' from the corresponding software sources. Please note that, `TrainTruecaser.py` expects the training corpus to be located in the `train.txt` file.
- Although Truecaser perhaps allows for better accuracy, its training script generates much larger models than those of Moses. This drastically increases the execution times so in production we suggest using `post_process_nltk.sh` with `<true_caser_type>` set to `moses`.

It remains to note that, the delivered demo, described in Section 3.2, has text pre- and post-processing enabled. So it can be used as a good example for establishing the production flow for the delivered infrastructure.

3.7 Empirical evaluation

This section describes an empirical comparison between the translation system developed within this project and the two others:

- **Oister** - developed by Dr. Christof Monz et al. at the University of Amsterdam, The Netherlands;
- **Moses** - a well established and well known system developed by Prof. Dr. Philipp Koehn et al. at The Johns Hopkins University, Baltimore, US;

All of these systems are implementing the phrase-based statistical (or data-driven) approach to machine translation (MT).

Since all of the systems listed above support multi-threaded decoding, the main purpose of our comparison is to determine how the systems' performance scales on a multi-core machine with the increasing number of threads. Please note that, in these experiments we do not employ the distributed infrastructure possibilities of the REMEDI project. I.e., we solely concentrate on the thread-related scalability of the systems and not the scalability related to using multiple translation servers with load balancing. In addition, we check on the systems' performance with respect to each other. The latter is done by comparing the translation times and the throughput. Let us remind the reader that, one of the main goals of the REMEDI project [Mon14] was to create a translation system that would be comparable in translations' quality but a number of times faster than its predecessor, namely Oister.

The rest of the section is organized as follows: Section 3.7.1 introduces the systems used in the experiments. Section 3.7.2 explains the experimental setup. Section 3.7.3 presents the obtained results and provides the analysis.

3.7.1 Systems

Let us introduce the systems used in the experimental comparison.

REMEDI: This is the system developed within the REMEDI project by Dr. Ivan S. Zapreev under the leadership of Dr. Christof Monz. The project is performed within the Information and Language Processing Systems group at the University of Amsterdam, The Netherlands.

This system is written mostly in C++ and supports multi-threading in the form of issuing a dedicated translation thread per source sentence. The employed Language, Translation and Reordering models are then shared in the multi-threaded environment between multiple sentences being translated.

Note that, REMEDI is made following a distributed client-server architecture. In our experiments a server is first started to load the models and prepare itself for performing translation tasks. Next a console client application is started. The latter sends the source text file, in batches, to the translation server and waits for all the translation to be received.

For our experiments we used the git snapshot *949dc64f493aac4a2aede6a56d6d6f2ecc5cac0a* of the system. The first system's git repository commit dates as early as *Fri Apr 17 13:15:36 2015 +0200*. This means that the system exists for about 2 years and it is therefore relatively immature.

Oister: This is a home-brewed system developed by Dr. Christof Monz et al. within the Information and Language Processing Systems group at the University of Amsterdam, The Netherlands.

This system is written mostly in Perl and supports multi-batching of the translated text corpus. The latter is done by splitting the text into a number of batches and translating each batch of sentences within a dedicated thread.

Oister does not provide a client-server architecture. It is a monolith system that, in order to do translations, each time has to load the required models. Moreover, Oister parallelizes translation process through batching: the sentences to be translated are split into a number of equally sized (in the number of sentences) chunks and then each chunk gets its translation thread to performance consequent sentences translation. This means that the translation time of Oister is always defined by the longest translation time among all the batches.

For our experiments we used the git snapshot *8e35f4a8bd3d54fe12b4b0aa36157248f34770ad* of the system. The first system's git repository commit dates as early as *Wed Sep 21 16:29:42 2011 +0200*. This means that the system exists for about 6 years now and this adds to its maturity.

Moses/Moses2: This is the system developed by Prof. Dr. Philipp Koehn et al. at The Johns Hopkins University, Baltimore, US;

Since 2016, Moses comes in two versions: The first one, we will keep calling Moses, is the evolutionary branch of the system. The second one, called Moses2, is a novel drop-in replacement for the Moses decoder. The latter is specifically designed to be fast and scalable on multi-core machines. The only information available for Moses2 at the moment is present online on the system's web page [Had16].

Both Moses and Moses2 are written mostly in C++. The reference paper for Moses, c.f. Section 3 of [Had10], states: "The threading model adopted for multi-threaded Moses assigns each sentence to a distinct thread so that each thread works on its own decoding task, but shares models with the other threads." Figure 2, shows how the translation times of Moses, stated in that paper, decrease with using more threads on a 16 core machine. The dependency is clearly not linear and is also not optimal as, e.g., going from 1 thread to 6 does not provide a 6 fold improvement but is just about 3.5 faster. More details can be found in Section 6 of [Had10].

In 2016 Moses2 was introduced, as specified in [Had16]: Moses2 is designed to be fast and scalable on multi-core machines. It only implements a subset of the functionality of Moses. Moses and Moses2 are not exactly alike, there's some differences in pruning, stack configuration etc. Figure 3 shows an experimental comparison of the throughput between Moses and Moses2.

It is important to point out, as this will be opposing our experimental results, that according to Figure 3, the extrapolated performance of Moses and Moses2 on a single thread should be virtually the same. However, the more threads are introduced the more distinct the difference becomes. We will come back this point later in Section 3.7.3 when discussing our experimental findings.

Both Moses and Moses2 can be run in a server mode but the implemented XML-RPC interface only allows to request a single sentence translation at a time and does not support asyn-

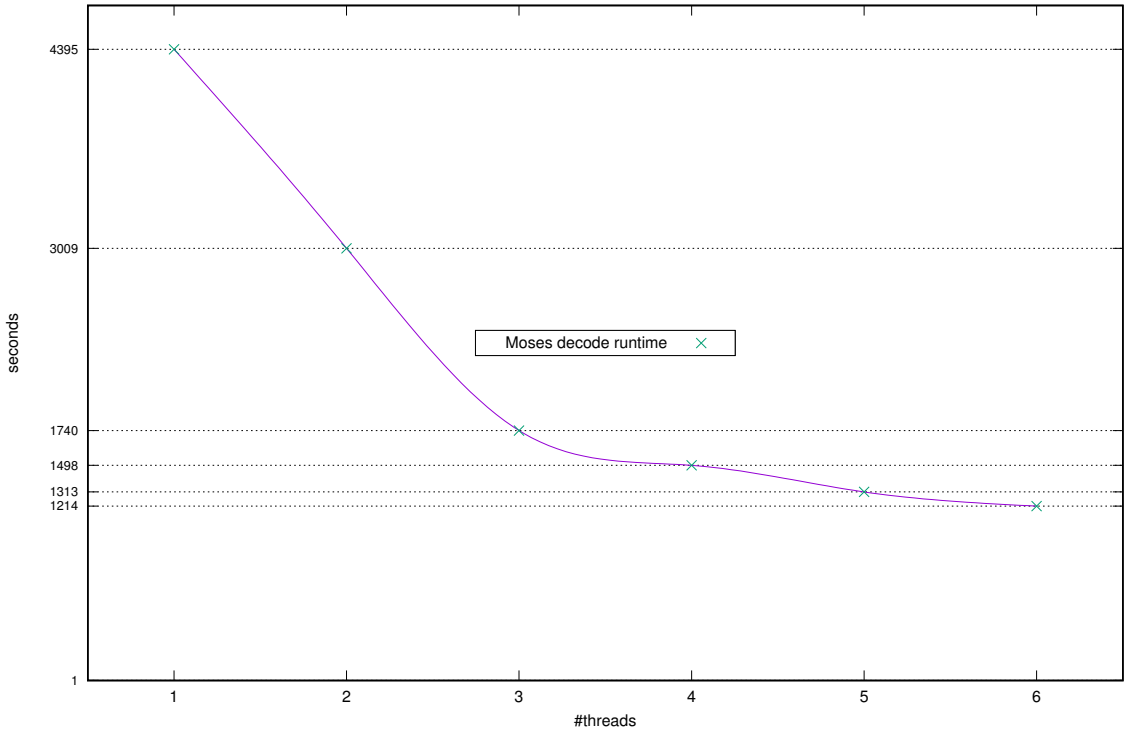


Figure 2: Moses scalability, 16 cores [Had10]

chronous calls to the Moses servers [2]. For a single client, this situation completely eliminates the advantage of multi-threaded server implementation. Clearly, before the client can request the next sentence translation, via the provided XML-RPC interface, it has to wait until the previous sentence is translated and the result is returned. Therefore to get the maximum performance out of Moses/Moses2, we used these systems in their stand-alone console mode. I.e. we ran them in the same way as we did with Oister.

For our experiments we used the git snapshot *0af59a4cda442adb9dd3b04542292c61f70bb504* of the system. The first system's git repository commit dates as early as *Mon Jul 3 18:10:46 2006 +0000*. This means that the system exists for about 11 years now and this adds to its maturity.

3.7.2 Setup

In order to measure performance of the aforementioned systems we chose to perform Chinese to English translations based on the data of the **MT-**04 dataset [Gro10]. Let us consider the experimental setup in more detail. We will first discuss the size of the used models, then go into the main translation parameters matching. Next, we indicate how we achieved the comparable BLEU performance of the systems. Finally, we explain how the decoding times were measured

[2]We considered all the provided Moses/Moses2 server client examples in Perl, Python and Java and none of them had multi-sentence or asynchronous calls implemented.

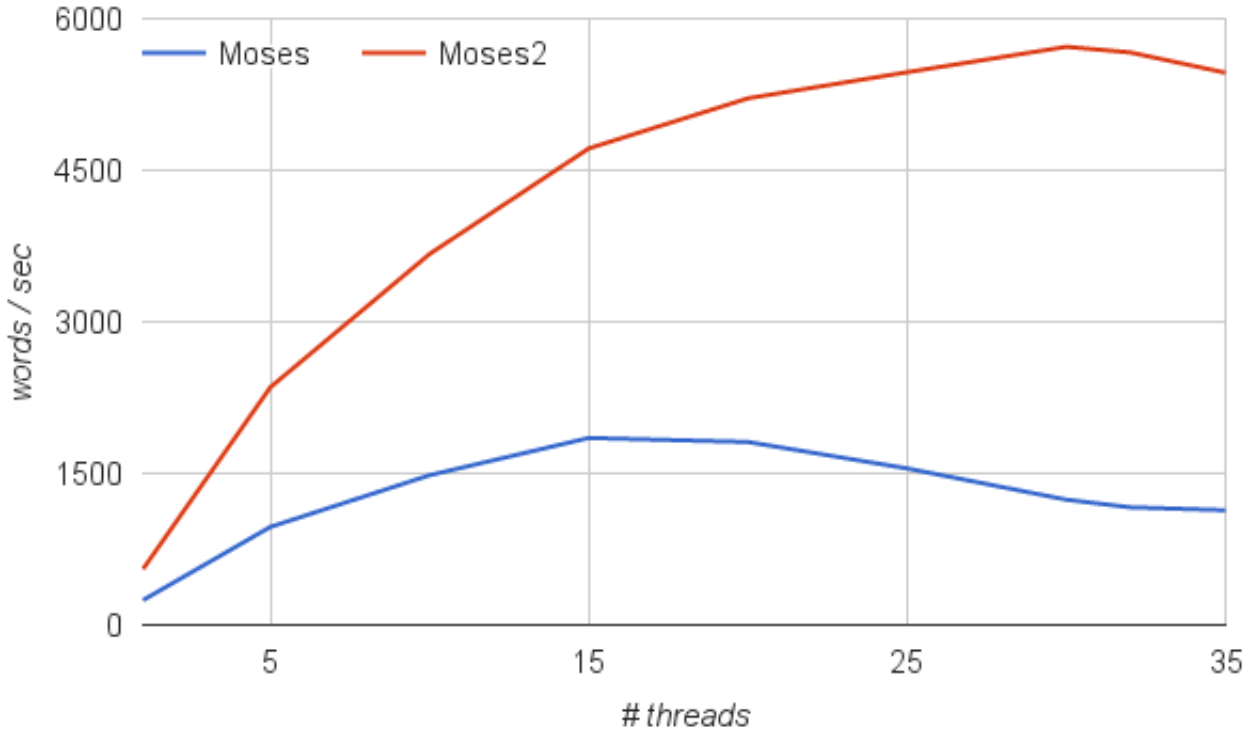


Figure 3: Moses vs Moses2, 32 cores [Had16]

and on which machine configuration the experiments were run.

Models: It has been decided to take large-sized models in order to make the timing aspects more vivid. The used model sizes are as follows:

- **Language Model** - 48.9 Gb (5-gram model);
- **Translation Model** - 1.3 Gb (5 features model);
- **Reordering Model** - 9.7 Gb (8 features model);

Parameters: We made sure that all of the system's parameters having high impact on systems' decoding times are matched in their values. We used:

- **Translation limit** $= 30$ - the number of top translations per source phrase to consider;
- **Beam width** $= 0.1$ - the maximum deviation from the best stack's hypothesis to consider;
- **Stack capacity** $= 100$ - the maximum number of hypothesis per translation stack;
- **Source phrase length** $= 7$ - the maximum source phrase length to consider;

- **Target phrase length** $= 7$ - the maximum target phrase length to consider;
- **Distortion limit** $= 5$ - the maximum distance for jumping within the source sentence;

Tuning: All of the systems were individually tuned on the **MT-**04 dataset [Gro10], using the same source corpus. We took a CTB segmented [MSB$^+$14] Chinese source text consisting of 1788 sentences and 49582 tokens. The same Chinese source text was used for translation during the performance experiments. The latter allowed us, in addition to performance measurements, to control the resulting translations' BLEU scores. Comparable BLEU values give us a higher confidence in proper performance comparison[3]. The BLEU scores of the resulting translations, per system, are listed below:

- REMEDI: **36.72** BLEU
- Oister: **36.80** BLEU
- Moses: **35.53** BLEU
- Moses2: **35.53** BLEU

Note that, both REMEDI and Oister have very close BLEU scores, the difference is 0.08 which is considered to be negligible. Moses and Moses2 show exactly the same scores, which comes at is a bit of a surprise as Moses and Moses2 are not exactly alike. Moses2 only implements a subset of the functionality of Moses, also there are some differences in pruning and stack configuration [Had16]. The last thing to note is that the scores difference between REMEDI/Oister and Moses/Moses2 is of 1.27 BLEU points. This 3.5% difference implies a non-ideal but rather fair performance comparison. Note that, this is the best what the systems could achieve on the given data as each of the systems was tuned independently to provide the best scores possible.

Measurements: All of the systems under consideration were taken as black-box systems. I.e., we did not rely on their timing outputs but rather measured the systems' run-time as given by the `time` command of the Linux shell. For each system we calculated the difference between the run-times needed to translate the Chinese source text, consisting of $1,788$ sentences and $49,582$ tokens, and a single Chinese word text. This difference gave us the pure translation times, excluding any model-loading/unloading, server-connection/disconnection or disk IO related times. Note that, each experiment was performed 10 times, which gave us the possibility to compute the average decoding times per experiment along with the standard deviation thereof. The exception was Oister: Due to its very long translation times, from 1.5 hours on 70 threads up to 20 hours on a single thread, we ran Oister complete-corpus translation experiments only three times. However, we did 10 runs of the single-word text translations to properly measure its model-loading and unloading times. Considering the drastic difference in the Oister runtime and that of other tools, the fact of fewer test run repetitions done for Oister should not have any impact on the obtained results.

[3]Assuming correct system implementations, low BLEU scores could be a sign of the decoding algorithm considering too few translation hypothesis

Machine: To conclude the experimental setup section, let us note that each experiment was run independently on a dedicated machine. The used test machine runs Cent OS 6 and features 256 Gb RAM and a 64-bit, 40 core Intel Xeon, 2.50 GHz processor. The complete machine configuration is given in Appendix A.

3.7.3 Results

In this section we present several plots obtained from the measured data. First, we will compare the systems' runtime and systems' throughput. Next, we consider systems' relative performance, scaling, and look at the speed-ups gained with increasing the number of decoding threads. At last, we investigate how REMEDI scales when increasing the workload. All of the plots presented in this section are based on the same data and just give different views on it for better analysis.

Systems' decoding times: Figure 4 shows four independent plots of the systems' decoding times. Note that the *x* and *y* scales are log_2 and log_{10} respectively. These plots show the average decoding time values plus the computed standard deviations. As one can see, Oister's deviations are rather small, compared to its large average decoding times. Next in line is REMEDI, its standard deviations are small, compared to those of Moses and especially Moses2 which indicates high application's stability in the multi-threading environment. This stability is also ensured by the fact that REMEDI is a client/server system, so the model loading/unloading times, requiring unstable disk I/O operations, are fully excluded from the measurements.

Figure 5 presents the systems' average decoding times next to each other. Here, one can make several observations: First of all, the performance of Moses and Moses2 are not like given in Figure 3 in Section 3.7.1. On a single thread, Moses is about 2 times slower than Moses2 and on $40 - 70$ threads it is about 1.5 times slower. The more threads the less difference there is between Moses and Moses2. This contradicts Figure 3, obtained from the official Moses2 web page [Had16], where the situation is quite different and suggests that Moses2 scales better than Moses solely due to a better multi-threading implementation. Our findings indicate that Moses2 actually scales worse than Moses with the number of threads. We speculate that currently the speed improvement of Moses2 over Moses is mostly gained by the improved decoding algorithms or used data structures and not multi-threading improvements. Second, when the #threads $\geq$ #cores the decoding times of REMEDI and Moses are about the same. We see it is a great achievement from our side, considering that REMEDI was developed within two years by a single developer and Moses is being developed for 11 years by a research community. Despite this great difference in development investments, when fully utilizing the machine cores both of these tools exhibit the same decoding performance.

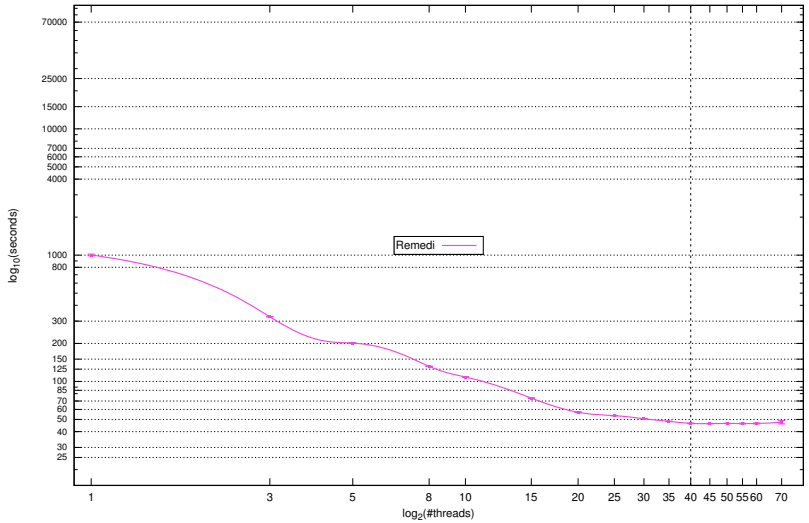
Systems' throughput: The average systems' throughput, in terms of words per second (wps), with standard deviation values, per number of threads, is given in Figure 6. These plots are functions of those in Figure 4 and give an insight into the wps data and its accuracy. Figure 7 shows the average throughput values for all of the systems, so that one can compare and investigate the wps performance. For example, REMEDI has ≈ 50 wps on a single thread which grows to ≈ 1000 wps on 40 threads. This indicates a ≈ 20 times performance increase when comparing

the system being run on a single core vs 40 cores. Going up in the number of threads beyond the number of available cores does not seem to bring any significant penalties.

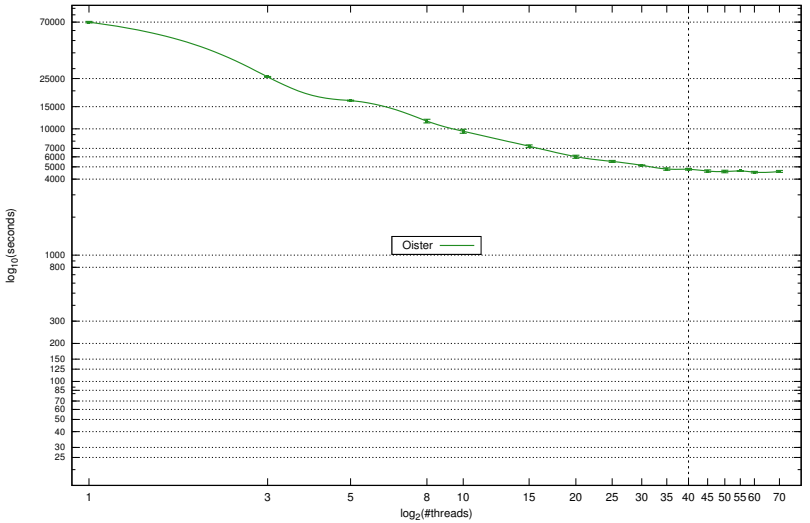
Relative systems' performance: One of the main project goals was to get a faster SMT system than Oister. Figure 8a gives the translation time ratio of Oister vs. REMEDI, with standard deviations. As one can see, on a single thread REMEDI is ≈ 70 times faster than Oister and on $25 - 40$ threads it is ≥ 100 times faster. The other plots show the systems' performance relative to the fastest tool: Moses2. An important thing here is that REMEDI is just ≈ 2.5 to ≈ 1.5 times slower than Moses2. Also this seems to be mostly due to more efficient decoding algorithms and data structures of Moses2 and not the efficiency of its multi-threading. The average decoding time ratio plots are given in Figure 9. This one shows that the difference between Moses and REMEDI is: *(i)* not that big; *(ii)* decreases with the increasing number of threads; *(iii)* is eliminated if #threads $\geq$ #cores.

Thread based scaling: One of the last but very important things to consider is the system's scaling factor with respect to its single threaded implementation. As before, Figure 10 gives four independent plots with standard deviations and Figure 11 provides the average-value plots of all the systems. As one can notice, the data of Moses and Moses2 is rather noisy, especially compared to that of REMEDI. Yet the results are representative and show clear trends. For example we see that Moses2 scales worst in the number of threads and a better scaling is exhibited by Oister that follows a batch-based parallelization strategy. A yet better system is Moses which goes up to 18 times efficiency on 30 cores. The best system is REMEDI, it shows clear and stable scalability until all of the cores are utilized. After that, the speed-up stays constant except for a small decline at 70 threads. However, looking at the increased standard deviation at the 70 threads point on Figure 10a, we suspect that it is just a statistical outlier.

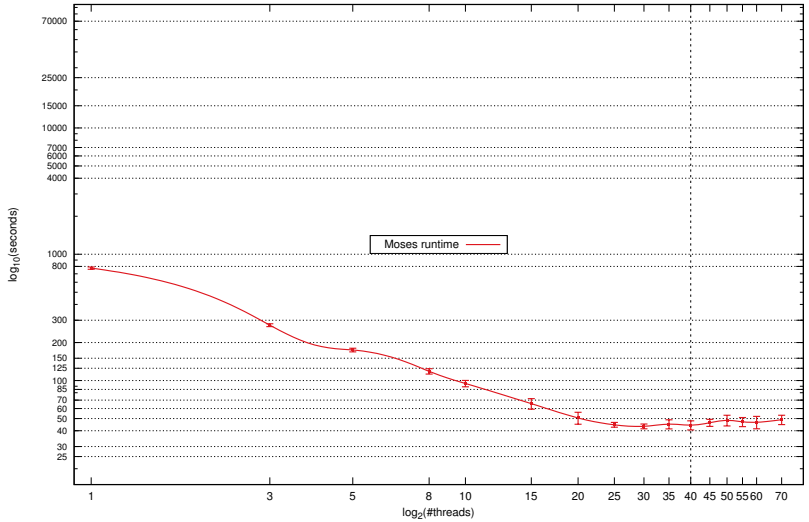
REMEDI load scaling: Figure 12 shows how the REMEDI performance scales with the number of input sentences. Here, we took the translation times for the original Chinese corpus of $1,788$ sentences and the translation times of the new corpus obtained by copying the original one twice, to get $3,576$ sentences. As one can see the translation times for each number of threads for the double corpus have simply doubled. This indicates linear scaling in the number of sentences between these two experiments. The latter is a good sign of scalability, meaning that the translation tasks scheduling is efficient.



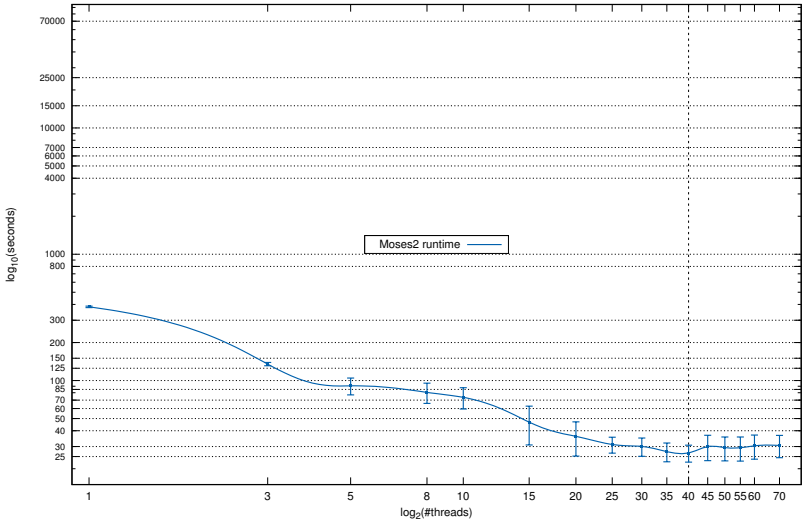
(a)



(b)



(c)



(d)

Figure 4: Decoding times, standard deviation

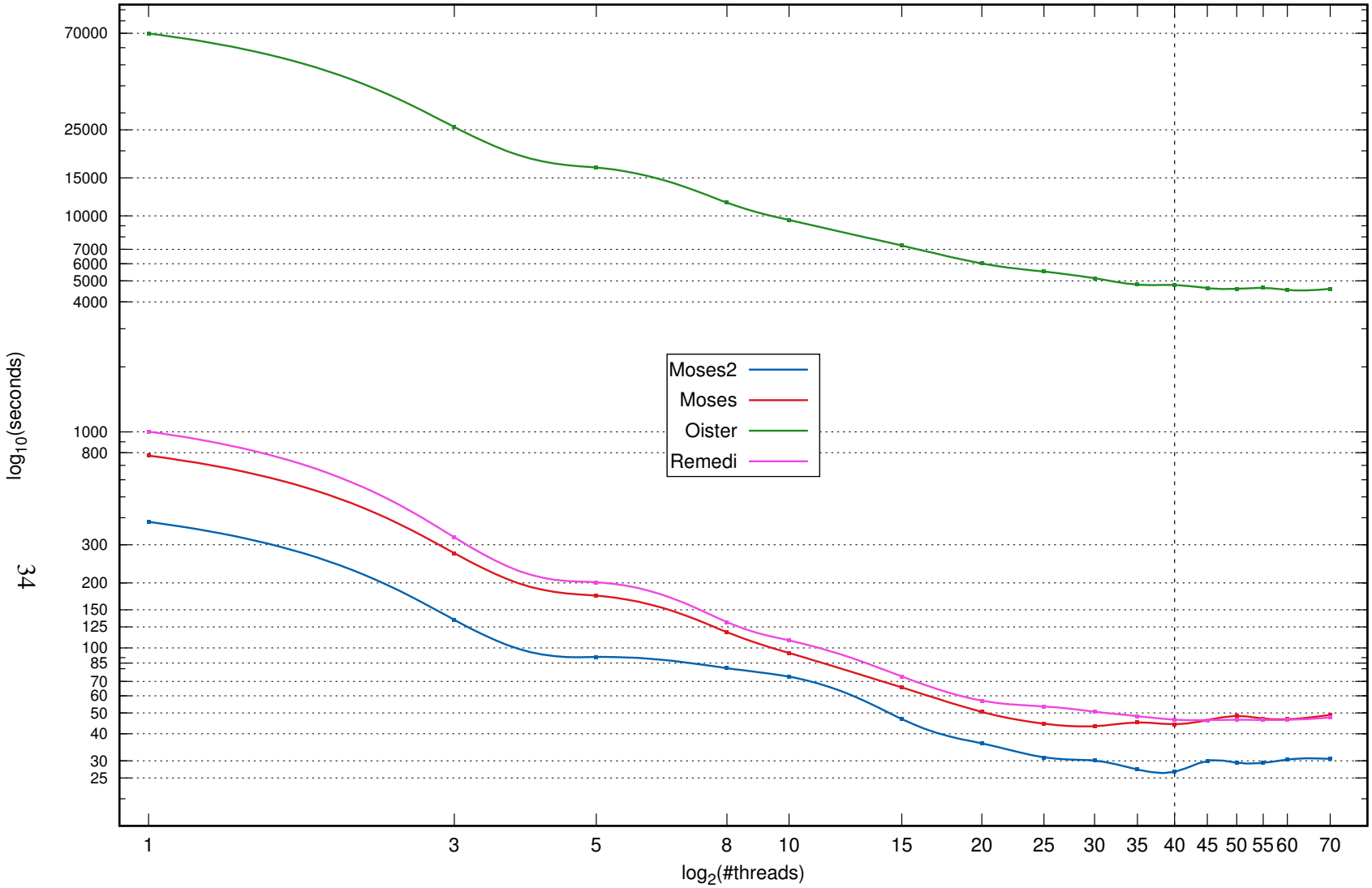
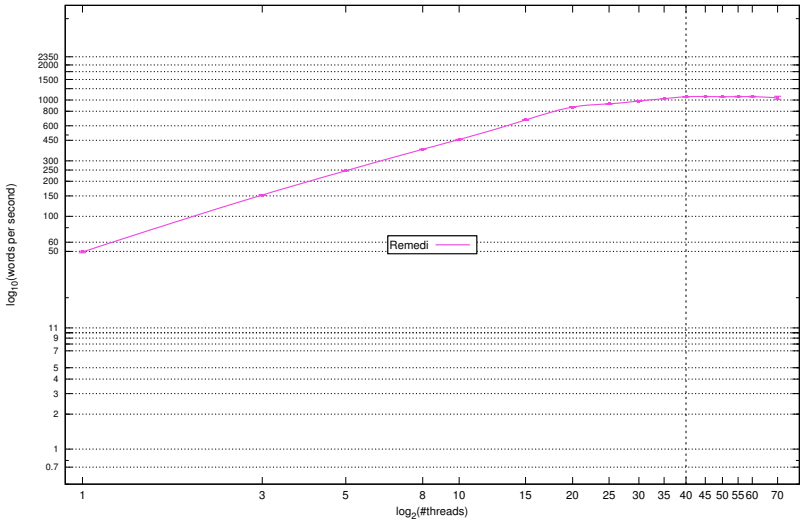
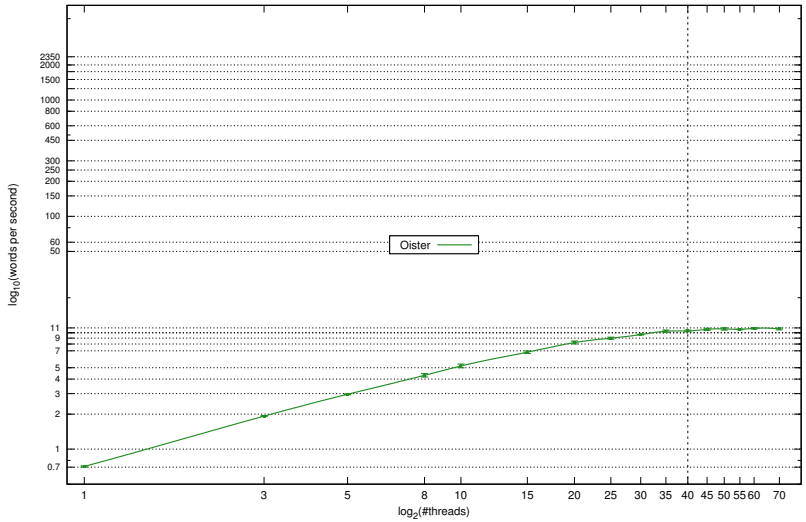


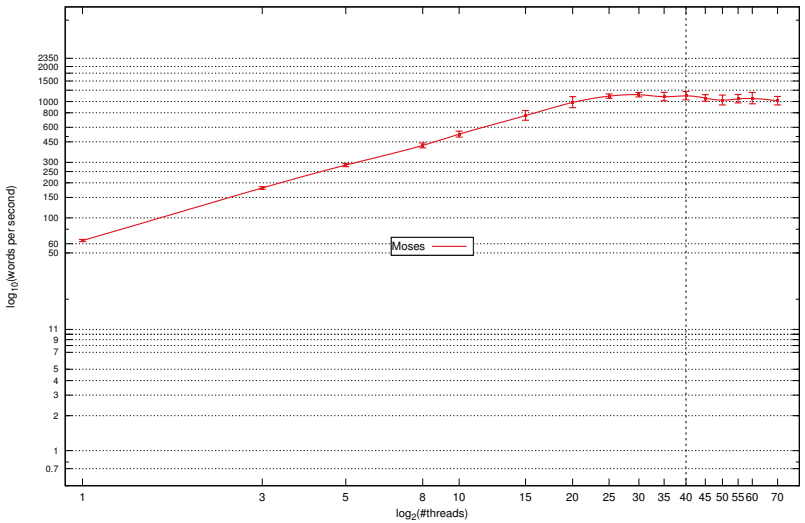
Figure 5: Decoding times



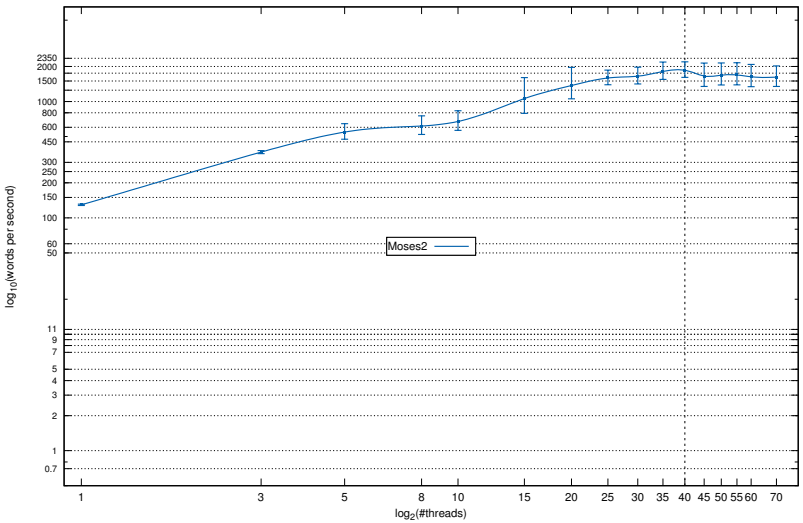
(a)



(b)



(c)



(d)

Figure 6: Words per second, standard deviation

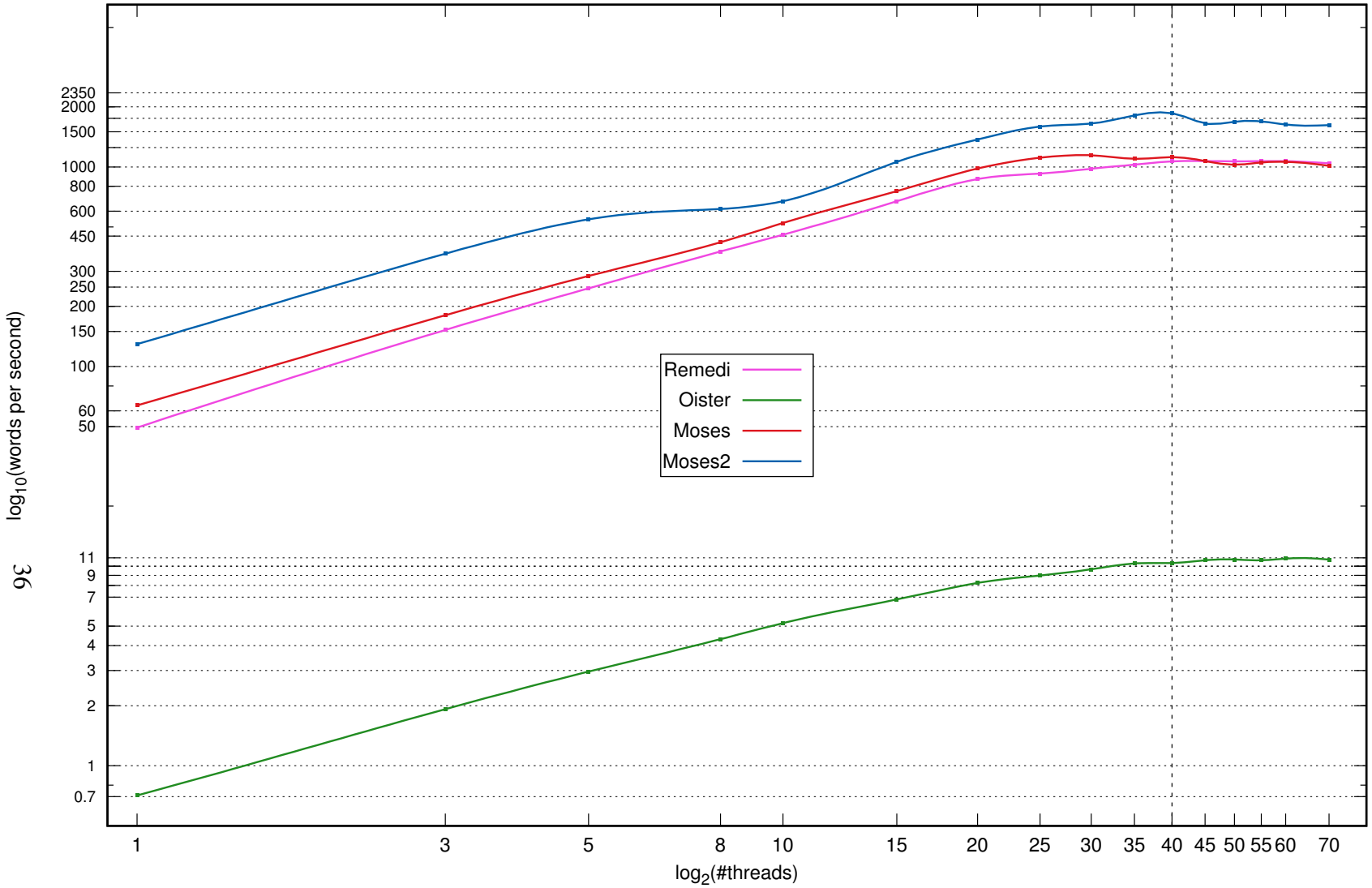
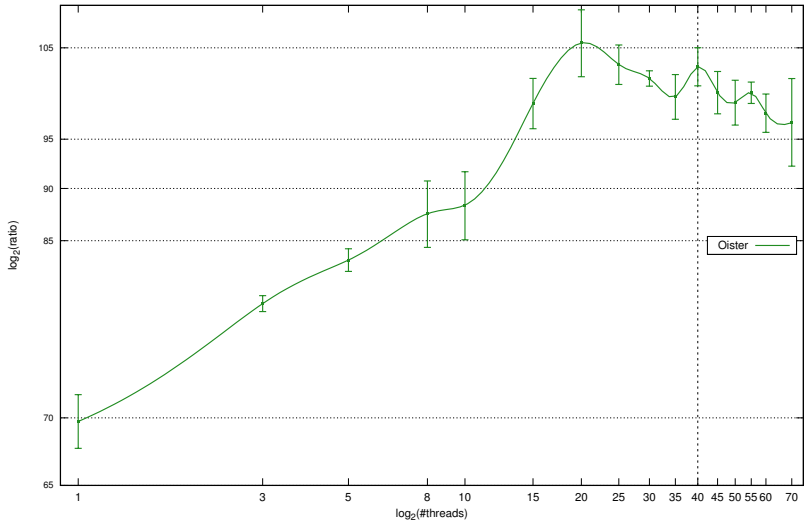
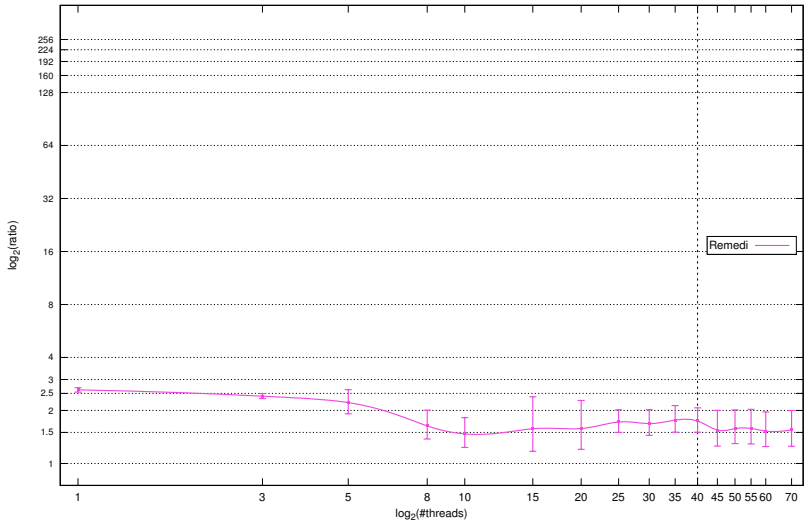


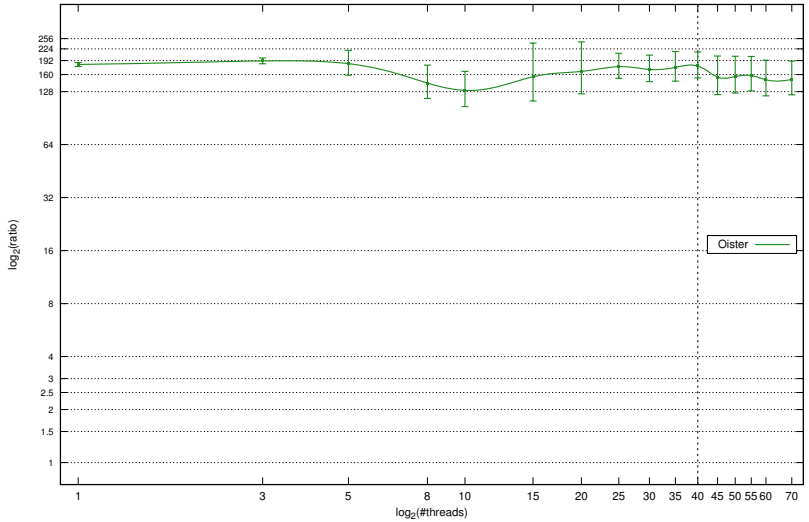
Figure 7: Words per second



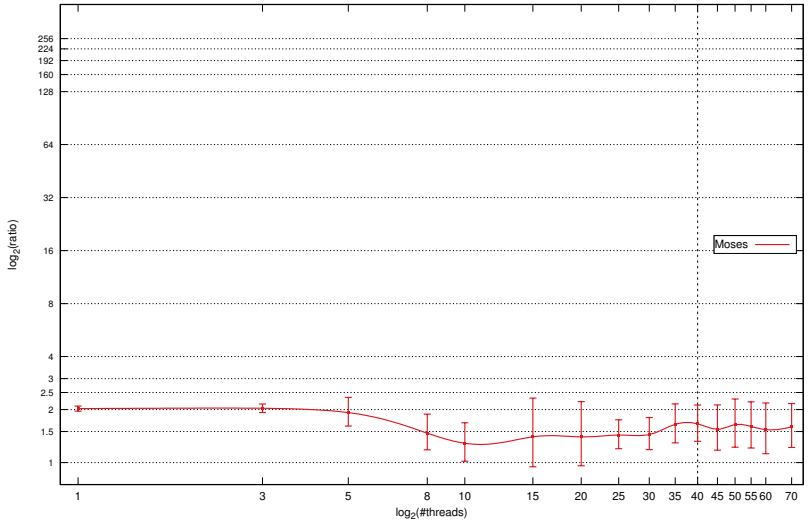
(a) Oister vs. REMEDI



(b) REMEDI vs. Moses2



(c) Oister vs. Moses2



(d) Moses vs. Moses2

Figure 8: Relative decoding times, standard deviation

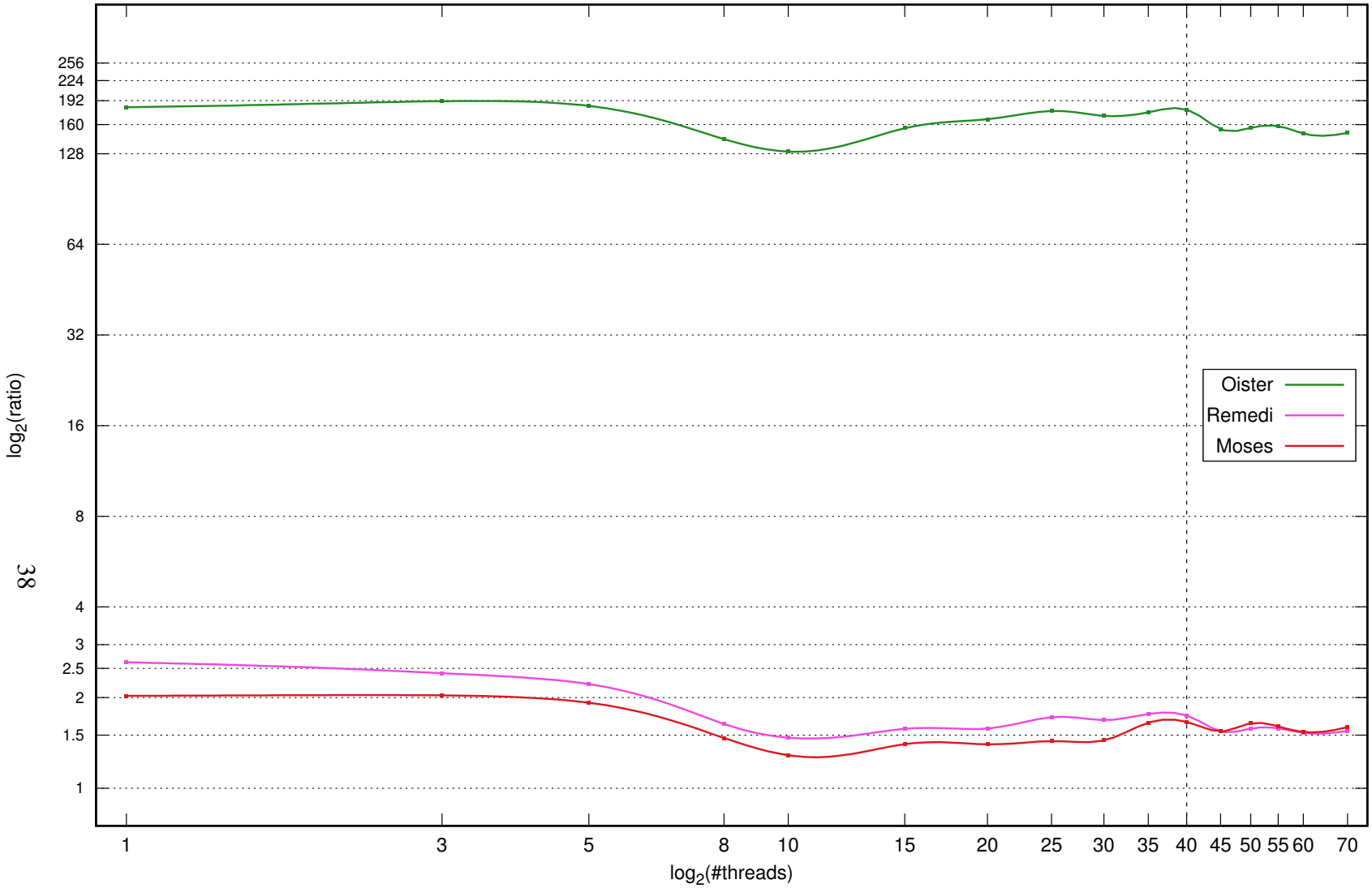
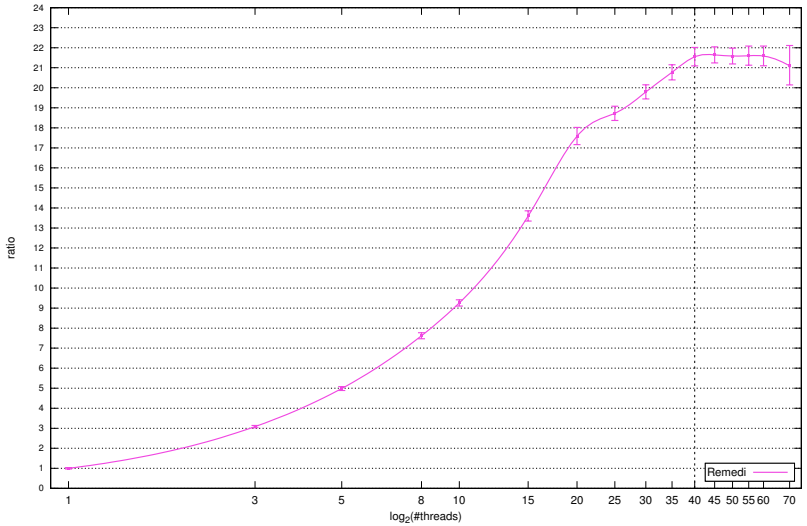
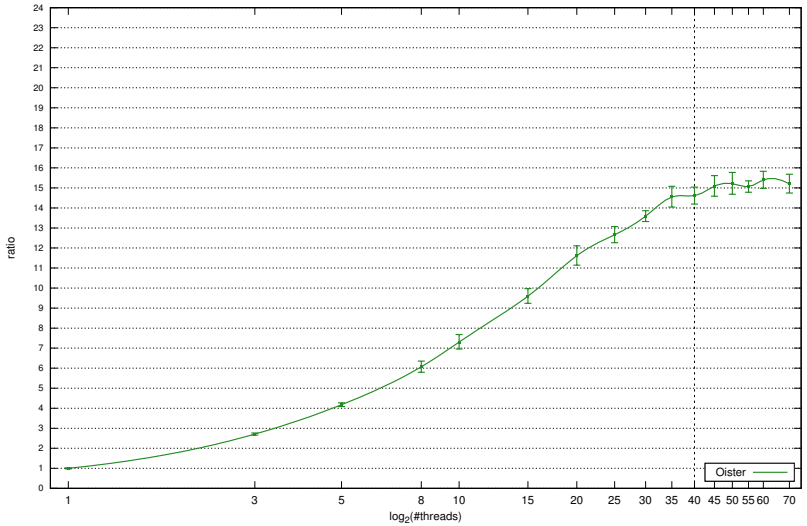


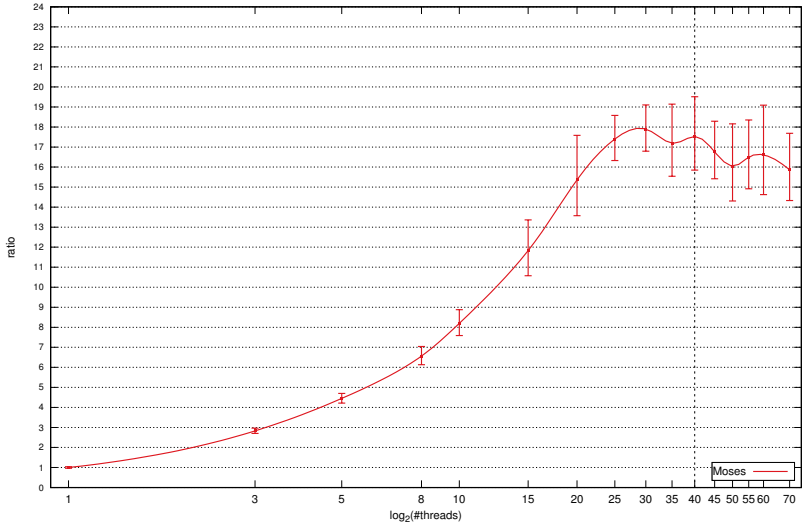
Figure 9: Decoding times relative to Moses2



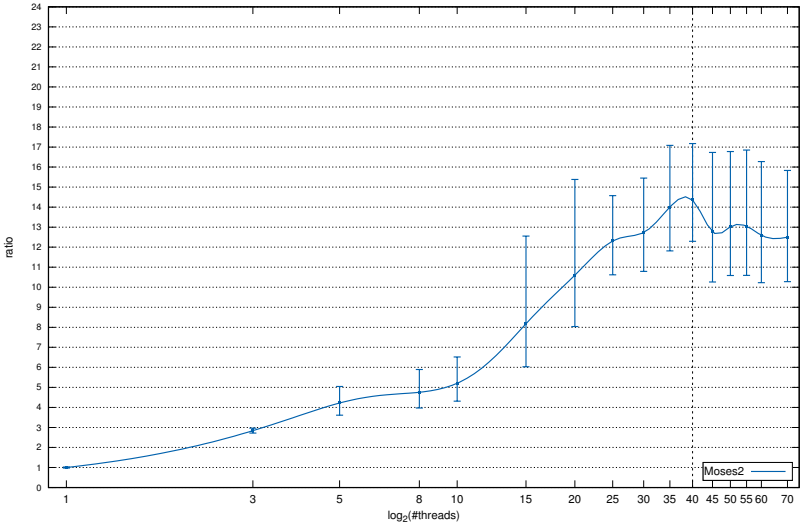
(a)



(b)



(c)



(d)

Figure 10: Speed-up relative to a single thread, standard deviation

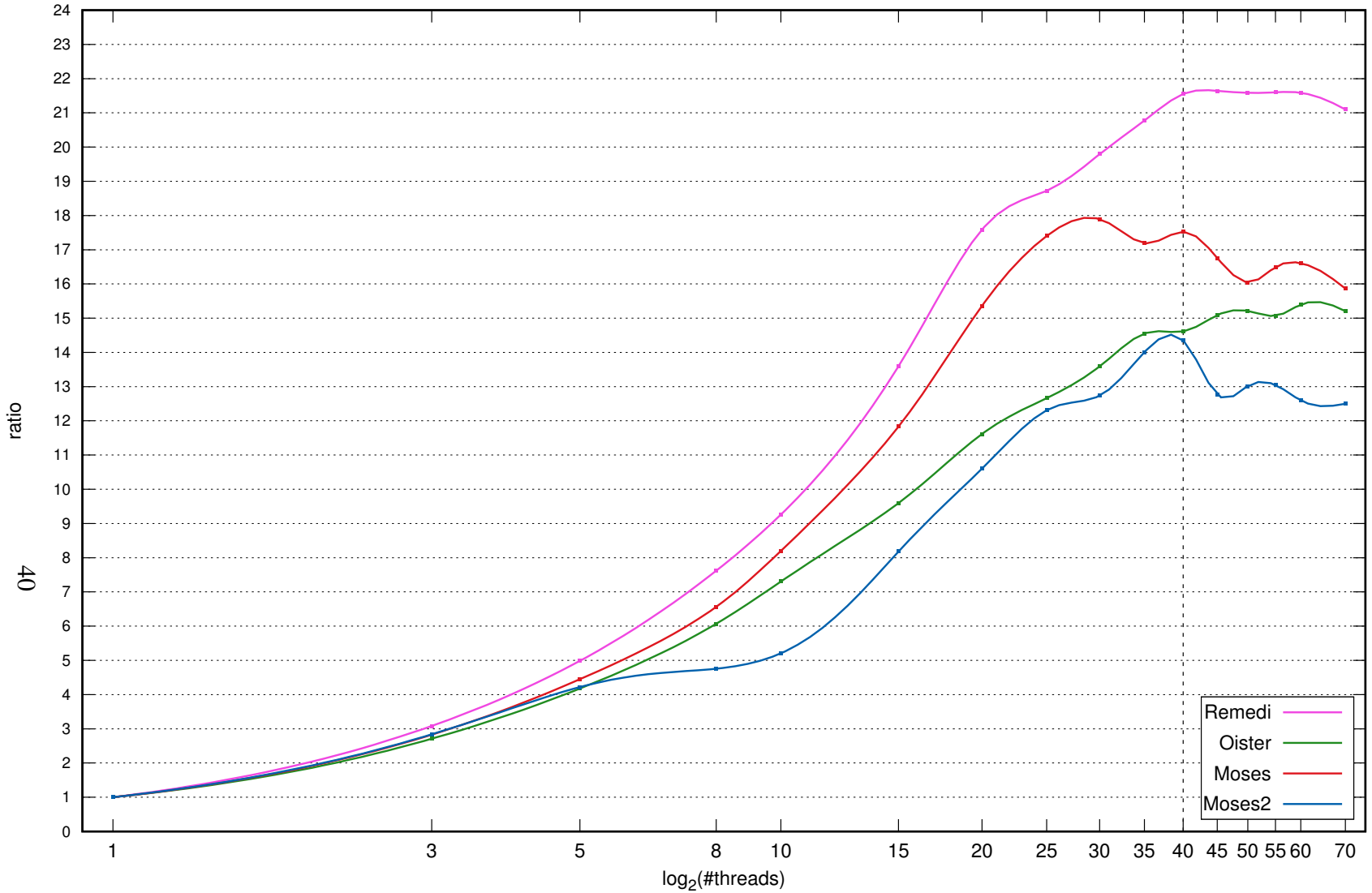


Figure 11: Speed-up relative to a single thread

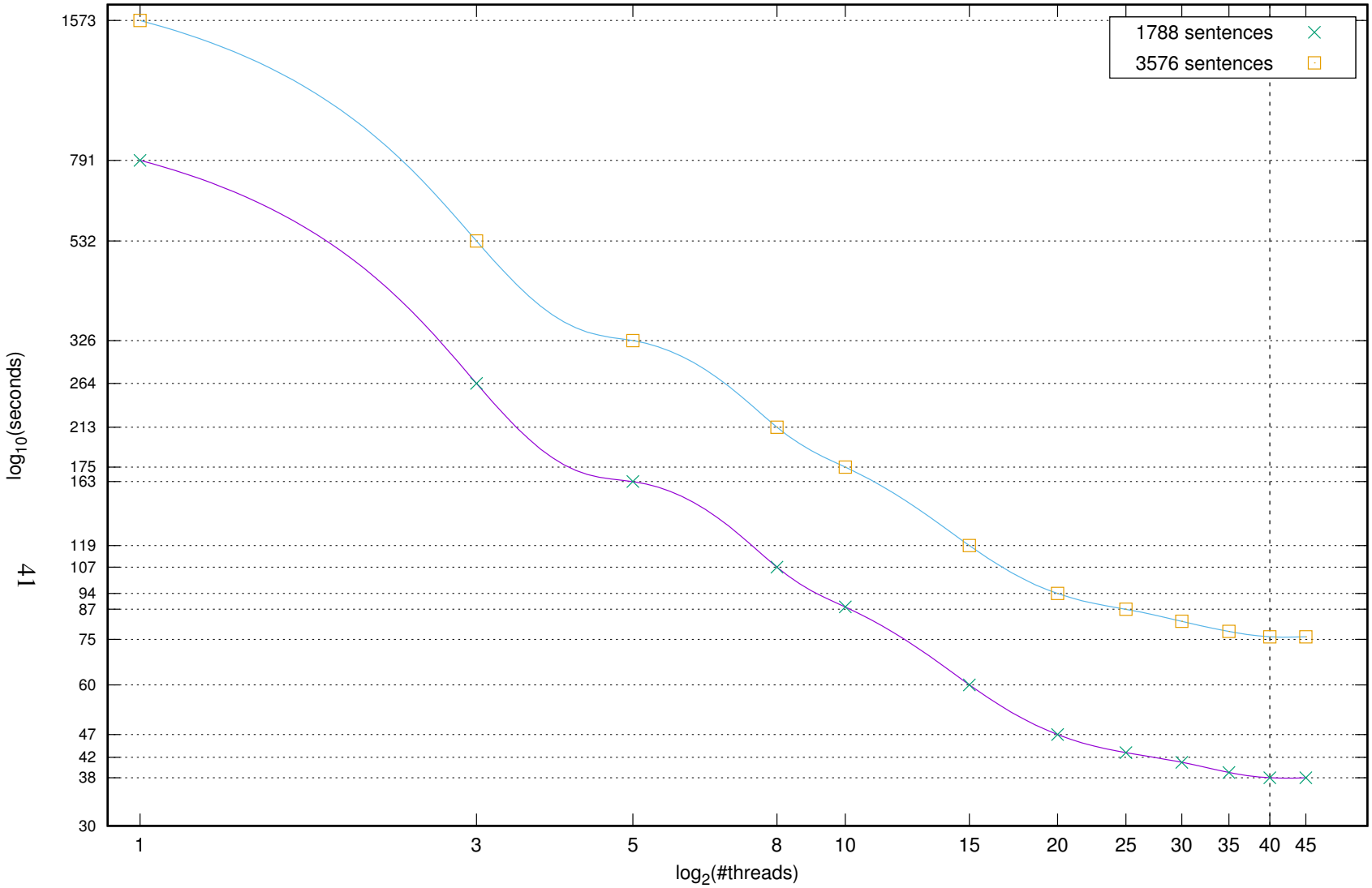


Figure 12: REMEDI decoding times, 1788 vs 3576 sentences

4 Conclusions

Let us summarize this report by listing the main items of the 24 month's REMEDI project delivery, with the corresponding short descriptions. This time we have supplied:

- **Communication protocols** - A thorough and complete description of the communication protocols between all the of system's applications to facilitate integration with third party products, c.f. Section 3.1;
- **Deliverable Demo** - Self-contained scripts, models and an instruction to run a demonstration of the system's infrastructure. This demo includes integration with third party tools for text pre/post-processing, c.f. Section 3.2;
- **Improved BLEU scores** - Software enhancements have resulted in a 30% BLEU score improvement of REMEDI on a full-fledged MT-04 Chinese to English test set. The REMEDI's performance now matches that of Oister and is higher than those of Moses and Moses2, c.f. Sections 3.3 and 3.7.2;
- **Integration with neural LM** - An experiment of integrating a neural Language Model into REMEDI, done by a fellow PhD student, under project member's guidance, has shown that extending our software is a straightforward and easy task, c.f. Section 3.4;
- **Tuning scripts** - Added scripts for models' parameter tuning make the decoding framework complete. Tuning the configuration file parameters guarantees the best quality[4] of translations provided by the system, c.f. Section 3.5;
- **Text processing scripts** - NLTK and Stanford Core NLP based scripts make the system convenient and user-friendly, c.f. Section 3.6. For text pre/post-processing they support:
  - Text lower/upper-casing: Multiple European languages;
  - Sentence splitting: Multiple European languages and Chinese;
  - Sentence (de-)tokenization: Multiple European languages and Chinese;
  - Language auto-detection: Multiple European languages and Chinese;
  - Target text structure restoration: All languages;
- **Empirical evaluation** - An extended experimental comparison between REMEDI, Oister, Moses and Moses2, c.f. Section 3.7, gives an outstanding correlation with the state of the art systems, from which it follows that REMEDI:
  - Has the best scaling capacity in the #threads;
  - Is very stable in its decoding times;
  - Is from 70 to 100 times faster than its predecessor: Oister;
  - Reaches Moses in decoding times if #threads $==$ #cores;
  - Is rather close to Moses2 in its decoding times;

[4] Measured in terms of the BLEU scores.

Please note that, this was the last period of an active system's development within the REMEDI project. Therefore the future work, to be done in the coming year, will be primarily limited by: Web client's user interface extensions; developing model building specifications; and writing additional project documentation, if required.

References

[AORP13] Tamchyna Aleš, Dušek Ondřej, Rudolf Rosa, and Pavel Pecina. Mtmonkey: A scalable infrastructure for a machine translation web service. *The Prague Bulletin of Mathematical Linguistics*, 100:31–40, October 2013. https://github.com/ufal/mtmonkey.

[COJA15] Truica Ciprian-Octavian, Velcin Julien, and Boicea Alexandru. Automatic language identification for romance languages using stop words and diacritics. *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 00:243–246, 2015.

[Gro10] NIST Multimodal Information Group. Nist 2004 open machine translation (openmt) evaluation, 2010. http://www.itl.nist.gov/iad/mig/tests/mt/2004/.

[Gru15] John Gruber. Markdown: A plain text formatting syntax. https://daringfireball.net/projects/markdown/, 2015.

[Had10] Barry Haddow. Adding multi-threaded decoding to moses. *Prague Bull. Math. Linguistics*, 93:57–66, 2010.

[Had16] Barry Haddow. Moses2: A drop-in replacement for the moses decoder., 2016. http://www.statmt.org/moses/?n=Site.Moses2.

[ISZ15] Christof Monz Ivan S. Zapreev. REMEDI: 6 month project report, 2015.

[ISZ16] Christof Monz Ivan S. Zapreev. REMEDI: 18 month project report, 2016.

[JAS06] Javascript object notation. https://www.ietf.org/rfc/rfc4627.txt, 2006. Accessed: 2016-09-06.

[KHB+07] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

[Koe10] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[LB02] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[LIRK03] Lucian Vlad Lita, Abe Ittycheriah, Salim Roukos, and Nanda Kambhatla. tRuEcasing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 152–159, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[Mon14] Christof Monz. Robust and efficient machine translation in a distributed infrastructure, 2014.

[MSB+14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Web11] The websocket protocol. https://www.rfc-editor.org/info/rfc6455, 2011. Accessed: 2016-09-02.

[Zap17] Ivan S. Zapreev. Readme.md: The software information document, 2017.

Appendices

A Test machine configurations (smt10)

The machine's CPU is:

```
[izapree1@smt7 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
```

```
  Stepping:              4
  CPU MHz:               1200.000
  BogoMIPS:              4999.27
  Virtualization:        VT-x
  L1d cache:             32K
  L1i cache:             32K
  L2 cache:              256K
  L3 cache:              25600K
  ...
```

The machine features 16 RAM modules:

```
  [izapree1@smt7 ~]$ dmidecode --type memory
...
Memory Device
Array Handle: 0x0029
Error Information Handle: Not Provided
Total Width: 72 bits
Data Width: 64 bits
Size: 16384 MB
Form Factor: DIMM
Set: None
Locator: P1-DIMMA1
Bank Locator: P0_Node0_Channel0_Dimm0
Type: DDR3
Type Detail: Registered (Buffered)
Speed: 1333 MHz
Manufacturer: Hynix Semiconductor
Serial Number: 0B6D55E2
Asset Tag: DimmA1_AssetTag
Part Number: HMT42GR7AFR4A-PB
Rank: 2
Configured Clock Speed: 1333 MHz
...
```