Divide and Conquer Strategy for Large Data MT

Dimitar Shterionov dimitars@kantanmt.com
KantanLabs, KantanMT, Dublin, Ireland

Abstract

In recent years Statistical Machine Translation (SMT) has established a dominant position among the variety of machine translation paradigms. Industrial Machine Translation computer systems, such as KantanMT, deliver fast and of high performance SMT solutions to the end user. KantanMT is a cloud-based platform that allows its users to build custom SMT engines and use them for translation via a batch or an online mode. In order to employ the full potential of the cloud we have developed an efficient method for asynchronous online translation. This method implements a producer-consumer technique that uses multiple queues as intermediate data storage units. Furthermore, each queue is associated with a priority that defines how quickly the queue can be consumed. That gives our users the control on the flow of translation requests, especially when it comes to large amounts of data.

In this paper we describe the design and the implementation of the new method and compare it to others. We then assess the improvement in the quality of service of our platform by empirical evaluation.

1 Introduction

In recent years Statistical Machine Translation (SMT) (Koehn (2010); Brown et al. (1993)) has established a dominant position among the variety of available machine translation paradigms. In 2007 Moses (Koehn et al. (2007)) was released – an open-source toolkit for SMT. While research efforts have been mainly focused on improving the core SMT technology, i.e., Moses and related pre- and post- processing techniques, we focused on bringing this technology to the end user in a highly scalable manner. Our MT platform, KantanMT[1] is fully distributed on the cloud. In order to employ the full potential of the cloud and provide to our users high quality translations for large amount of data with low response time, we have developed an efficient request handling system.

The system aims to optimize resource allocation and to improve the robustness and resilience of our platform as well as the quality of service (QoS). We analyse two approaches for processing translation requests in a distributed MT environment that are already employed in our online translation pipeline and compare them to our new method.

The first approach processes each request at the moment it has been received using a centralized *https* endpoint. According to this method the segment, sent via the request, is translated by a predefined SMT model and returned back to the user. Underlying is a load balancing mechanism to distribute the segments on a fleet of servers that are dedicated for translation. This method is synchronous (we refer to it as *SYNC*) and thus can introduce high delays between sending a request and receiving a translation.

Our second method implements an efficient producer-consumer technique based on a single intermediate data storage unit (IDSU). While it allows users to translate large amounts of

[1]https://kantanmt.com/

segments asynchronously (thus we refer to it as *ASYNC*), it is still bound by the limits of the used IDSU. According to the *ASYNC* method, old requests need to be processed before new ones. Thus, it may cause delays for new requests.

The new method we have developed improves upon these restrictions by distributing requests in as many IDSUs as the user requests. It allows requests that arrive at different time points to be processed in *parallel* and *independently* from each other. Furthermore, this method adds an extra control layer that allows our users to assign priority to their requests – the higher the priority, the faster the requests are processed. That is, we take under consideration that some requests may be more important to our users than others, and as such would need to be handled faster. We refer to this method as the *PP-ASYNC*.

In this paper we describe the design and the implementation of the *PP-ASYNC* method and compare it to the other two. We then assess the improvement in the QoS of our platform by empirical evaluation.

2 Online translation with KantanMT

KantanMT (`https://kantanmt.com`) is a cloud-based SMT platform that provides machine translation services to its clients for more than 760 language pairs. It is based on the state-of-the-art Moses toolkit to train SMT models; these models are then used for decoding. In the remaining of this paper we use the term *KantanMT* or *SMT "engine"* to refer to the collection of SMT models and configuration files. The use of the Moses toolkit together with the distributed architecture of the system allows KantanMT engines to be built at very high speed and with low computational cost.

KantanMT platform is equipped with two translation modes – *batch* mode and *API* or *online* mode. In the batch mode clients provide a set of documents[2]. The system then translates each of these files (one after the other) and returns their translated versions to the client.

The focus of this work is on the methods used in online translation mode. The online translation mode allows clients to send translation requests via the KantanMT API using an HTTP GET or an HTTP POST method. Each request specifies user identification (under the form of a KantanAPI token), the engine to be used for translation and the segment to be translated. In response to a given translation request, the user will receive a translation of the input segment. In the rest of this section we describe the three online translation methods. We focus on the *PP-ASYNC* method which is the most recent and innovative online translation method of KantanMT.

2.1 Synchronous online translation

SYNC is the most basic online translation method of KantanMT. According to this method the user sends a request for translation of one (or more) segments via the API call `translate` and receives either the translation of the segment or a failure notification. In the request (both HTTP GET and HTTP POST requests are supported) the user provides its unique API token, the machine translation (MT) profile to be used for translation and, finally, the segment. Example 2.1 shows an HTTP GET request for the SYNC method.

Example 2.1 *An HTTP GET request from a user with token 1234567890123456[3] for translation of the segment* "Welcome to our blog '100% Machine Translation'." *with the MT profile* MT-en-bg.

https://kantanmt.com/api/translate/1234567890123456/MT-en-bg/Welcome to our blog '100%25 Machine Translation'. △

[2]KantanMT supports 26 file formats.

[3]This is an example token.

After using the token to verify the user, KantanMT translates the segment(s) using the specified MT profile (if such exists and is running). The response of the `translate` call to each translation request is either a successful translation or an error notification (in case a failure due to failed authentication, incorrect encoding, etc. has occurred).

In Figure 1 we show the architecture of the system for handling SYNC requests with respect to message-passing, i.e., the flow of information from the user to the platform and back. Although we refer to this method as synchronous, it has an underlying distributed architecture that allows several requests to be processed in parallel (i.e., several segments to be translated in parallel). This we achieve by using two load balancers (LB) – one to receive and distribute the requests and a second one that distributes the segment for translation among different machines.

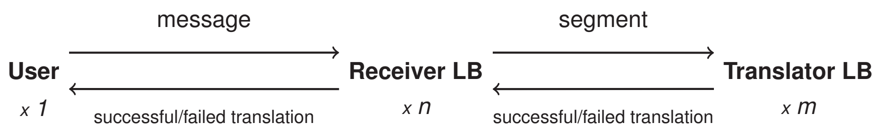


Figure 1: Message passing in SYNC.

2.2 Asynchronous online translation

Despite the capabilities of the LBs in the SYNC method to parallelise requests, thus letting multiple segments to be translated simultaneously, users still need to wait until one batch of segments is translated before they can submit new ones. This has three main drawbacks: (i) lower effectiveness of our platform – i.e., lower quality of service – enforcing users to manually operate the submission process; (ii) a large amount of requests submitted at the same time may overload the LBs and cause huge latencies and (iii) concurrent users may not be able to process their requests in parallel.

In order to tackle the aforementioned problems we introduce an additional layer of parallelism that allows users to submit huge batches of messages[4]. This method uses a queue as an intermediate data storage unit and processes translation requests asynchronously; we refer to this method as the *ASYNC* method.

The architecture of the *ASYNC* system implements a producer/consumer-based (Arpaci-Dusseau and Arpaci-Dusseau (2015)) approach where incoming requests are stored in a queue. A *consumer* then depopulates the queue sending each read segment for translation. To translate a segment, we use the SYNC method. If the translation succeeds it is forwarded to the user in the form of a notification sent to a user-defined endpoint. If the translation fails, the segment is redriven for a new translation attempt; if the number of failed attempts exceeds a predefined threshold a failure notification is sent to the user on a user-defined endpoint. We present the architecture of the *ASYNC* system in Figure 2.

3 Parallel priority-based online translation

The *ASYNC* method allows users to deal with large amount of requests asynchronously and receive response from our system when a translation is generated, or when a failure occurs. However, often a user may decide to, e.g., delay some translations in order to have other segments processed sooner, e.g., messages for a job that is close to a deadline need to be processed faster than others that are not urgent. When it comes to large volumes of data prioritization can be crucial for the on-time service delivery.

[4]In practice, we do not set any restriction on the number of requests that can be submitted at once.

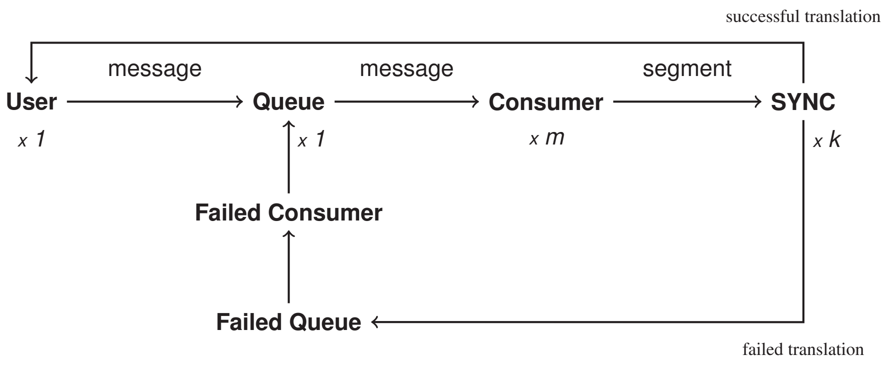


Figure 2: Message passing in *ASYNC*.

The *ASYNC* method is bound by the limits of the used data structure, i.e. the queue – each message is processed in the order that it is pushed into the queue: old messages already in the queue need to be processed before new ones. Such a scenario is given in Example 3.1.

Example 3.1 *Consider that User A sends* 2000000 *requests at time* T_0*. Next, consider that at time* $T_1 = T_0 + 10\ minutes$ 10000 *requests have been processed. That is, the* consumption ratio *is* 1000 *requests/minute. At time* T_1 *User B sends* 10000 *requests (stored at the end of the queue). While the* 10000 *requests of User B would require* 10 *minutes for processing – i.e., to translate the segments they carry – this user will need to wait for* $1990\ minutes = 33\ hoursand10\ minutes$ *before they can be processed.* △

3.1 Method description

In order to provide to our users a mechanism to control the priority of their requests as well as to enforce parallelism we devised the *Parallel priority-based* ASYNC, (PP-ASYNC). The *PP-ASYNC* method uses multiple queues, each associated with a priority and a name. Each incoming translation request contains the priority with which it should be processed and the name of the queue in which it should be stored.

Each queue has a priority and is consumed at a predefined ratio associated with that priority. The consumption ratio defines how quickly a queue is read. The higher the queue priority, the greater the consumption ratio. Once a requests is read from a queue the segment that it carries is sent for translation with the SYNC method.

Successful translations are then sent back to the user (via a success notification endpoint provided by the user). Failed translations are redriven into the correct priority queue for new translation attempt; if the number of translation attempts exceeds a predefined threshold then the segment is failed and the user is notified of the failure (via a failed notification endpoint).

The implementation of the *PP-ASYNC* method extends the implementation of the *ASYNC* method with: (i) a distributing mechanism that processes incoming requests and redirects them to the queue that matches the specified priority and queue name; if such a queue does not exist it is created before the request is pushed; and (ii) support for reading messages from multiple. Figure 3 shows the architecture of the system and the information flow.

3.2 Consumption ratio

The distributor (see Figure 3) receives a request from the user and distributes it to the correct queue according to the specified priority and queue name. The consumer (independently from the distributor) iterates over all queues, identifies the consumption ratio of each queue, based

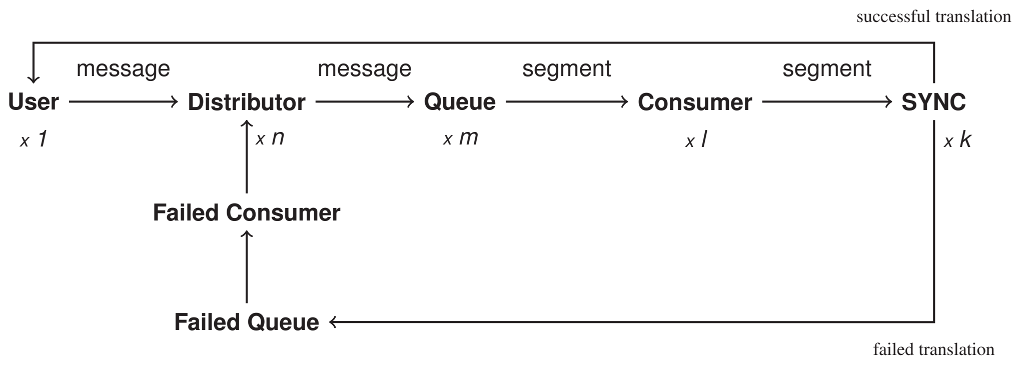


Figure 3: Message passing in PP-ASYNC.

on its priority, reads a number of requests from the queue, sends the segments they carry for translation and proceeds to the next queue. In our definition of the *PP-ASYNC* method, the consumption ratio of a queue defines how many requests are read from that queue at one iteration of the queue consumer. That is, for a queue $Q_i \in \{Q_1..Q_N\}$ with priority q_i there will be c_i requests read. For another queue $Q_j, j > i$ there will be $c_j > c_i$[5] requests that need to be read.

Our implementation allows multiple distributors and consumers to run simultaneously as neither of them has side effects. That is why the actual consumption ratio depends on the consumption ratio c_i of a queue Q_i and the number of consumers (M). For example, if there are two consumers that read simultaneously from a queue with consumption ratio 5 (messages per iteration), then ten requests will be read from that queue. A more detailed example is given in Example 3.2.

Example 3.2 *Consider that User A sends* 2000000 *requests to a queue with priority* 1 *(*Q_1*) at time* T_0 *and that there are no other queues at that time*[6]*. Assume that the consumption ratio for priority* 1 *is* 100 *requests per iteration and the consumption ratio for priority* 2 *is* 500*. Also, consider that in* 1 *minute one consumer can read* 100 *requests and there are* 10 *consumers running simultaneously. Then, in* 10 *minutes, that is, at time* $T_1 = T_0 + 10\, minutes$ 10000 *requests will have been processed from* Q_1*. At time* T_1 *User B has just send* 10000 *requests all of which with priority* 2*, i.e., they are stored in a queue with priority* 2 *–* Q_2*. The consumers become aware of the new queue and proceed to processing it. According to its priority each consumer will process* 500 *requests (in one iteration). Given that each consumer reads from a queue at the speed of* 100 *requests per minutes, at time* $T_2 = T_0 + 5\, minutes$ *the consumers will stop reading from* Q_2 *and start a new iteration. They will consume* 1000 *requests from* Q_1 *and at time* $T_3 = T_2 + 1\, minute$ *the consumers will move to* Q_2*. At time* $T_4 = T_3 + 5\, minutes$ *all requests from* Q_2 *will be read, that is, approximately* 11 *minutes after they have been submitted.* $\triangle$

To determine the optimal consumption ratio we empirically evaluated different values and compared the system's performance. Our tests showed that the most efficient consumption ratio is the one linear to the priority of the queue. In our implementation we have selected the simplest linear dependency: the consumption ratio equals the priority of the queue. That is, for

[5]If the number of requests that needs to be read from a queue is larger than the number of requests available in the queue, then all of them are read, the segments they carry are sent for translation, and the consumer proceeds to the next queue.

[6]Or all other queues are empty.

a queue Q_i with priority q_i in one iteration of a handler there will be q_i requests read from that queue. We aimed at a balanced consumption ratio where, on the one hand, queues with higher priority are consumed faster than queues of lower priority but on the other hand, lower-priority queues are still processed within reasonable time.

In order to avoid concurrences between different users we allow two queues with the same priority to have different names. In this way, each user can benefit equally from the PP-ASYNC. In addition, we have implemented a queue with *infinity* priority – consumers process messages from this queue until it is empty. Using this queue leads to a exactly the same behaviour as the *ASYNC* method.

4 Empirical evaluation

We performed a series of tests that aimed to empirically evaluate the new online translation method, i.e., the *PP-ASYNC* method. There are two objectives that we targeted with our test: (i) compare the performance (i.e., translation speed) of the *ASYNC* and the *PP-ASYNC* methods and (ii) show that the new method eliminates/reduces delays for newly incoming messages.

We ran our experiments on a Windows Server 2012 machine with 8-core Intel CPU, 15GB or RAM and 160GB of SSD. For all our tests we used 2 engines – E_1 and E_2 – of different size[7]. Engine E_1 is considered *large* as it is trained on approximately 108000000 words; engine E_2 is trained on approximately 500000 words and we refer to it as small.

4.1 Delays for new messages in the *PP-ASYNC* method

This experiment aims to reveal whether the *PP-ASYNC* method reduces the delay for processing new requests. We performed 5 tests. For each of them we used 10000 requests randomly selected from an English text. We also used 100 consumers. Specific details about the tests are shown in Table 1.

Test	Queues	Details
PP-ASYNC sequential	– 5 priority queues $Q_1..Q_5$. Queue $Q_i \in \{Q_1,..,Q_5\}$ has consumption ratio i.	– 1000 translation requests for engine E_1 are first stored in each queue. – 1000 translation requests for engine E_2 are stored in each queue afterwards.
PP-ASYNC parallel	– 5 priority queues $Q_1..Q_5$. Queue $Q_i \in \{Q_1,..,Q_5\}$ has consumption ratio i.	– 1000 translation requests for engine E_1 and 1000 translation requests for engine E_2 are in a random order.
PP-ASYNC distributed	– 5 priority queues $Q_1^{E_1}..Q_5^{E_1}$. Queue $Q_i^{E_1} \in \{QE_{11},..,QE_{15}\}$ has consumption ratio i. – 5 priority queues $Q_1^{E_2}..Q_5^{E_2}$. Queue $Q_i^{E_2} \in \{QE_{21},..,QE_{25}\}$ has consumption ratio i. Total: 10 queues.	– 1000 translation requests for engine E_1 in each queu $Q_i^{E_1}$. – 1000 translation requests for engine E_2 in each queu $Q_i^{E_2}$.
ASYNC sequential	– 1 queue	– 5000 translation requests for engine E_1 are first stored in the queue and 5000 translation requests for engine E_2 are stored in the queue afterwards.
ASYNC parallel	– 1 queue	– 5000 translation requests for engine E_1 and 5000 translation requests for E_2 are stored in a random order.

Table 1: Tests for determining the effect of the *PP-ASYNC* method on request delays.

We ran each experiment and measured the time when a request is consumed from a queue. We then measure the time difference between the first processed request (at overall) and the

[7]We use the term “engine size” to refer to the number of words which were used to train the engine.

	Delay time (in minute)				
Test name	E_1		E_2		Total
	First	Last	First	Last	
PP-ASYNC sequential	0.00	30.29	7.61	38.17	38.17
ASYNC sequential	0.00	22.65	20.98	44.90	44.90
PP-ASYNC parallel	0.00	34.14	0.02	34.13	34.14
ASYNC parallel	0.00	28.93	0.02	25.58	28.93
PP-ASYNC distributed	0.00	33.34	0.46	31.85	33.34

Table 2: Time until the first message for a given engine is read from a queue.

first processed request for a specific engine. That is, we compute the delay before the requests to a specific engine are accessed for the first time. For example, a given queue has requests for engine E_1 and engine E_2. The first request to be read will have no delay (0.00 minutes). Let us say that it is from engine E_1. 10 minutes after that a consumer reads the first request for engine E_2 resulting in a delay of 10 minutes for engine.

Our results for each of the five tests are summarized in Table 2.

From Table 2 we notice, first that for the parallel tests there is no difference between the *ASYNC* and PP-SYNC methods. That is because the requests are randomly distributed and, in practice, they are not prioritized.

The results for the *PP-ASYNC* sequential, *ASYNC* sequential and *PP-ASYNC* distributed are of greater interest as they show the benefits of using the *PP-ASYNC* method. Namely, from the comparison of the *PP-ASYNC* sequential and the *ASYNC* sequential we notice that using multiple queues decreases the delay significantly. As for the *PP-ASYNC distributed* test the delay is reduced even more, shown that using separate queues to store the requests for different engines is a prefered option when it comes to quick system response. In Figure 4 we compare the delays represented as a percentage of the total processing time.

Figure 4 again shows that the *PP-ASYNC* method with multiple queues designated to one engine has the best performance, i.e., almost no delay[8]. On large scale, i.e., when users send huge volumes of translation requests, such delay reductions may be crucial for their operation.

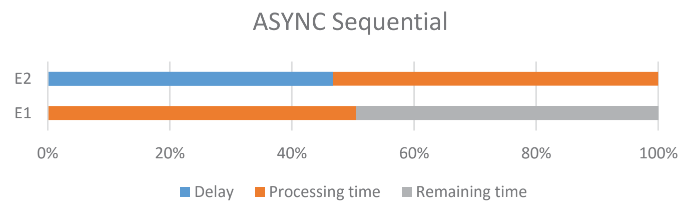
4.2 Comparison between *ASYNC* and PP-ASYNC

As shown in Section 4.1 the worst case scenario for the *PP-ASYNC* method is when messages are randomly distributed among different queues (see Table 2). Then there is no practical difference between the *PP-ASYNC* and the *ASYNC* method. We therefore compare the two methods in such a scenario in order to test whether there is a degradation in performance due to the additional distribution mechanism of the *PP-ASYNC* method. For this experiment we used 50000 requests for 2 engines and invoked 100 consumers. The two tests we executed are described in Table 3.

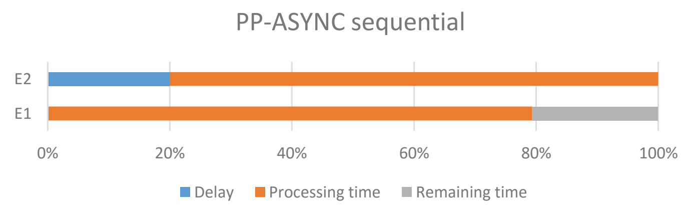
We ran three iterations for each test and we measured the time consumed from the moment the first request is read from a queue until it is emptied. We show the results from these tests in Table 4 in minutes.

The results summarised in Table 4 confirm that in the worst case scenario for the *PP-ASYNC* method the performance of the online translation of KantanMT does not degrade.

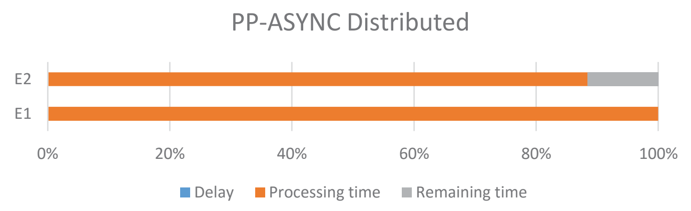
[8]In practice the delay is approximately one second.



a. The *ASYNC* sequential test.



b. The *PP-ASYNC* sequential test.



c. The *PP-ASYNC* distributed test.

Figure 4: Comparison of delays and processing times, relative to the total time.

Test	Queues	Details
PP-ASYNC comparison	– 6 priority queues $Q_{-1}, Q_1..Q_5$. Queue $Q_i \in \{Q_1, .., Q_5\}$ has consumption ratio i. Q_{-1} has infinite consumption ratio.	– 25000 translation requests for engine E_1 and 25000 translation requests for engine E_2 are in a random order.
ASYNC comparison	– 1 queue	– 25000 translation requests for engine E_1 and 25000 translation requests for E_2 are stored in a random order.

Table 3: Tests for comparing the *PP-ASYNC* and *ASYNC* in the worst case scenario.

5 Conclusions

In this paper we presented the online translation methods of KantanMT. We focused on our most innovative method – the PP-ASYNC. It uses multiple intermediate data storage units to store users requests; efficient producer-consumer technique is used to distribute the requests efficiently for translation. Each intermediate data storage unit is implemented as a queue and is assigned a prioirity. The priority defines at what rate the queue is consumed.

The new method extends the *ASYNC* method by introducing multiple queues and priorities for each queue. By using priorities this method allows our users to move forward the translation of important segments. Such a mechanism is crucial to our users, especially when it comes to

Test name	Consumption ratio	Iteration 1	Iteration 2	Iteration 3	Average
PP-ASYNC comparison	infinite	17.09	27.91	67.58	37.53
	1	57.84	55.38	85.75	66.32
	2	49.31	46.62	75.68	57.20
	3	44.91	42.71	73.22	53.61
	4	39.56	37.20	65.49	47.42
	5	36.07	30.09	60.88	42.35
Total		72.93	75.66	114.73	87.78
ASYNC comparison	-	83.24	72.80	110.29	88.78

Table 4: Consumption time (in minutes).

processing large volumes of data.

Our empirical evaluation showed that while in the worse case scenario the *PP-ASYNC* is as efficient as the *ASYNC* method, in general it reduces delays drastically. As such, our method is suitable for processing concurrently/in parallel requests from different users. Furthermore, the distributed architecture of our platform allows the *PP-ASYNC* method to handle large amounts of data efficiently and respond on time.

In the future we aim at optimizing this method by building an intelligent system to enforce extra control on the sysetm with less human interventions.

References

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2015). *Operating systems: Three easy pieces*. Arpaci-Dusseau.

Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311.

Koehn, P. (2010). *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi Itc-Irst, N., Cowan, B., Shen, W., Moran Mit, C., Zens, R., Aachen, R., Dyer, C., Constantin, A., College, W., and Cornell, E. H. (2007). Moses: Open source toolkit for statistical machine translation. pages 177–180.