Fast, Scalable Phrase-Based SMT Decoding

Anonymous ACL submission

Abstract

The utilization of statistical machine translation (SMT) has grown enormously over the last decade, many using open-source software developed by the NLP community. As commercial use has increased, there is need that for the software to be optimized for their requirements, in particular, faster phrase-based decoding and more efficient utilization of modern multicore servers.

In this paper we re-examining the major components of phrase-based decoding and decoder implementation with particular emphasis on speed and scalability on multicore machines. The result is a drop-in replacement for the Moses decoder which is up to fifteen times faster and scales monotonically with the number of cores.

1 Introduction

SMT has steadily progressed from a research discipline to commercial viability during the past decade as can be seen from services such as the Google and Microsoft Translation services. As well as general purpose services such as these, there is a large number of companies that offer customized translation systems, as well as companies and organization that implement in-house solutions. Many of these customized solutions use Moses as their SMT engine.

For many users, decoding is the most time-critical part of the translation process. Making use of the multiple cores that are now ubiquitous in todays servers is a common strategy to ameliorate this issue. However, it has been noticed that the Moses decoder, amongst others, is unable to efficiently use multiple cores (Fernández et al., 2016). That is, decoding speed does not substantially increase when more cores are used, in fact, it may actually *decrease* when using more cores. There have be speculation on the causes of the inefficiency as well as potential remedies.

This paper is the first we know of that focuses on improving decoding speed on multicore servers. We will take a holistic approach to solving this issue, re-implementing the decoder's structure to optimize for speed, re-assessing some of the optimization assumptions made in the implementation of significant parts of the decoder, as well as exploring changes to the core search algorithm. We will compare and contrast decoder implementations, not just decoding algorithms or individual components.

We will present a phrase-based decoder that is significantly faster than the Moses baseline for single-threaded operation, and scales monotonically with the number of cores.

As far as possible, model scores and functionality are compatible with Moses to aid comparison and ease transition for existing users. All source code will be made available under an open-source license.

1.1 Prior Work

Most prior work on increasing decoding speed look to optimizing specific components of the decoder or the decoding algorithm.

Heafield (2011) and Yasuhara et al. (2013) describes fast, efficient datastructures for language models. Zens and Ney (2007) describes an implementation of a phrase-table for an SMT decoder that is loaded on demand, reducing the initial loading time and memory requirements. Junczys-Dowmunt (2012) extends this by compressing the on-disk phrase table and lexicalized re-ordering model.

Chiang (2007) describes the cube-pruning and cube-growing algorithm which allows the trade-off between speed and translation quality to the adjusted with a single parameter. Wuebker et al. (2012) note that language model querying is amongst the most expensive operation in decoding. They sought to improved decoding speed by caching score computations early pruning of translation options. This work is similar to Heafield et al. (2014) which group hypotheses with identical language model context and incrementally expand them, reducing LM querying.

Fernández et al. (2016) was concerned with multi-core speed but treat the decoding process as a black box within a parallelization framework.

There are a number of phrase-based decoding implementations, many of which implements the phrase-based extensions above.

The most well known is Moses (Koehn et al., 2007) which implements a number of speed optimzations, including cube-pruning. It is widely used for MT research and commercial use.

Joshua (Li et al., 2009) also supports cube-pruning for phrase-based models. Phrasal (Spence Green and Manning, 2014) supports a number of variants of the phrase-based model. Jane (Peitz et al., 2012) supports the language model look-ahead described in Wuebker et al. (2012) in addition to other tools to speed up decoding such as having a separate fast, lightweight decoder. mtplz is a specialized decoder developed to implement the incremental decoding described in Heafield et al. (2014).

The Moses, Joshua and Phrasal decoders implement multithreading, however, they all report scalability problems, either in the paper (Phrasal) or via social media (Moses[1] and Joshua[2]).

Jane and mtplz are single-threaded decoders, relying on external applications to parallelize operations. This has the advantage it is easier to parallelize processing across multiple servers, not just multiple cores. As we shall see, this method does not scale beyond a small number of cores when attempting to scale to multiple cores within one server.

This paper focuses on overcoming the shortcomings of existing decoding implementations on multicore servers.

The rest of the paper will be broken up into the following sections. Next, we will describe the phrase-based model and the major implementation components, with particular emphasis on decoding speed. We will then describe modifications to improve decoding speed and present results. We conclude in the last section and discuss future work.

2 Phrase-Based Model

The objective of decoding is to find the target translation with the maximum probability, given a source sentence. That is, for a source sentence s, the objective is to find a target translation $\hat{t}$ which has the highest conditional probability $p(t|s)$. Formally, this is written as:

$$\hat{t} = \arg\max_t p(t|s) \quad (1)$$

where the *arg max* function is the search. The log-linear model generalizes Equation 1 to include more component models and weighting each model according to the contribution of each model to the total probability.

$$p(t|s) = \frac{1}{Z} \exp(\sum_m \lambda_m h_m(t,s)) \quad (2)$$

where λ_m is the weight, and h_m is the feature function, or 'score', for model m. Z is the partition function which can be ignored for optimization.

2.1 Beam Search

A translation of a source sentence is created by applying a series of translation rules which together translate each source word once, and only once. Each partial translation is known as a *hypothesis*, which is created by applying a rule to an existing hypothesis. This process is called *hypothesis expansion* and starts with a hypothesis that has translated no source word and ends with a completed hypotheses that have translated all source words. The highest-scoring completed hypothesis, according to the model score, is returned as most probable translation, $\hat{t}$.

In the phrase-based model, each rule translates a contiguous sequence of source words but successive translation options do not have to be adjacent on the source side, depending on the distortion limit. The target output is constructed strictly left-to-right from the target side of successive translation rules.

[1] https://github.com/moses-smt/mosesdecoder/issues/39

[2] https://twitter.com/ApacheJoshua/status/342022794097340416

A beam search algorithm is used to create the completed hypothesis set efficiently. Partial translations are organized into stacks where each stack holds a number of comparable hypotheses. Most phrase-based implementations place hypotheses in the same stack that have the same coverage cardinality $|C|$, where C is the coverage set, $C \subseteq \{1, 2, ...|s|\}$ of the number of source words translated.

2.2 Feature Functions

Features functions are the h_m in Equation 2, calculating a score for each hypothesis.

The standard feature functions in the phrase-based model include:

1. log transforms translation model probabilities, $p_{TM}(t|s)$ and $p_{TM}(s|t)$, and word-based translation probabilities $p_w(t|s)$ and $p_w(s|t)$,
2. log transforms of the lexicalized re-ordering probabilities,
3. log transforms of the target language model probability $p(t)$,
4. a distortion penalty
5. a phrase-penalty,
6. a word penalty,
7. an unknown word penalty.

The first three feature functions frequently trained on data and require the feature to read the model from files. The other feature functions are simple and do not leave much room for optimization. The language model consumes a large part of decoding time, and it is for this reason that it has already been heavily optimized.

2.3 Translation Model

For any realistic sized phrase-based model to be used in an online situation, memory and loading time constraints requires us to use load-on-demand phrase-table implementations. Moses contains a number of such implementations with different performance characterics, we show the time take to decode 800,000 sentences for the fastest two in Figure 1. From this, it appears that the Probing phrase-table (Bogoychev, 2013) has the fastest translation rule lookup, especially with large number of cores, therefore, we will concentrate exclusively on this implementation in our work as this.

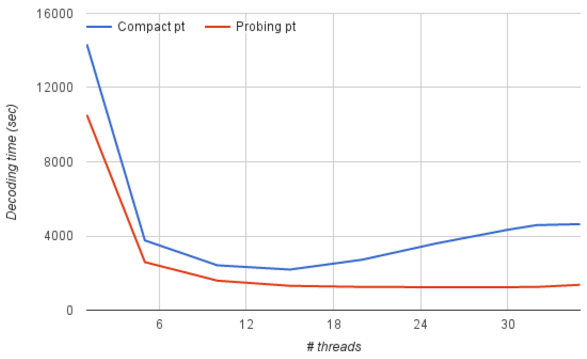

Figure 1: Moses decoding time with two different phrase-table implementations

2.4 Lexicalized Reordering Model

The lexicalized reordering model is trained on parallel data, usually requiring random lookups of the model file during decoding. Using the model can increase translation quality at the cost of longer decoding time. Efforts such as Junczys-Dowmunt (2012) have been made to increase the speed of the model.

3 Proposed Improvements

There are four main optimizations that we shall investigate. Firstly, the search creates and destroy a large number of hypothesis objects in memory which puts a burden on the operating system. We shall optimize the search algorithm to use memory pools and object pools, replacing the operating system's general purpose memory management with our own application-aware management.

Secondly, we shall investigate different stack configurations other than coverage cardinality to see whether they can improve speed and translation quality.

Thirdly, we will look at for optimizating the phrase-table, specifically in relation to the Probing implementation, focusing on two techniques. One, we will investigate caching of often used translation rules. The default caching framework save the most recently looked up rules. However, this caching mechanism actually makes decoding slower, Table 1. We shall explore a simpler

	No cache	Caching
Decoding time	2032	2302 (+13%)

Table 1: Decoding time (in sec with 32 threads) when using phrase-table cache

caching mechanism that creates a static cache at the start of decoding.

Two, the Probing phrase-table use a simple compression algorithm to compress the target side of the translation rule. Compression was championed by Junczys-Dowmunt (2012) as the main reason behind the speed of their phrase-table but as we saw in Figure 1, this comes at the cost of scalability to large number of threads. We shall therefore take the opposite approach to and explore optimizing decoding speed by disabling compression.

Lastly, the lexicalized reordering model require random lookups but query key are the source and target phrase of each translation rule. Rather than storing this model separately, we shall investigate integrating it into the translation model.

4 Experimental Setup

We trained a phrase-based system using the Moses toolkit with standard settings. The training data consisted of most of the publicly available Arabic-English data from Opus (Tiedemann, 2012) containing over 69 million parallel sentences, and tuned on a held out set. The phrase-table was then pruned, keeping only the top 100 entries per source phrase, according to $p(t|s)$. All models files were then binarized; the language models were binarized using KenLM (Heafield, 2011), the phrase table using the Probing phrase-table, lexicalized reordering model using the compact datastructure. These binary formats were choosen for their best-in-class multithreaded performance. Table 2 gives details of the resultant sizes of the model files. For testing decoding speed, we used a subset of the training data, Table 3.

	ar-en	fr-en
Phrase table	17	5.8
Language model	3.1	1.8
Lex re. model	2.3	637MB

Table 2: Model sizes in GB

For verification with a different dataset, we also used a second system trained on the French-English Europarl corpus (2m parallel sentences). The two different systems have characterics that we are interested in analyzing; ar-en have short sentences with large models while fr-en have overly long sentences with smaller models. Where we need to compare model scores, we used held out test sets.

	ar-en	fr-en
For speed testing		
Set name	Subset of training data	
# sentences	800k	200k
# words	5.8m	5.9m
Avg words/sent	7.3	29.7
For model score testing		
Set name	OpenSubtitles	newstest2011
# sentences	2000	3003
# words	14,620	86,162
Avg words/sent	7.3	28.7

Table 3: Test sets

Standard Moses phrase-based configurations are used, except that we use the cube-pruning algorithm (Chiang, 2007) with a pop-limit of 400, rather than the basic phrase-based algorithm. The cube-pruning algorithm is often employed by users who require fast decoding as it gives them the ability to trade speed with translation quality via a simple pop-limit parameter.

As a baseline, we use the latest version of the Moses decoder taken from the github repository.

For all experiments, we used a Dell PowerEdge R620 server with 16 cores, 32 hyper-threads, split over 2 physical processors (Intel Xeon E5-2650 @ 2.00GHz). The server has 380GB RAM. The operating system was Ubuntu 14.04, the code was compiled with gcc 4.8.4 and Boost 1.59[3] and the tcmalloc library[4].

5 Results

5.1 Optimizing Memory

We create a dynamic memory pool which can grow as more memory is requested. The memory is not released, instead the pools are reset and re-used. We instantiate two pools for each thread, one which is never reset and another which is reset after the decoding of each sentence. Datastructures are created in either pool according to their life cycle.

For critical datastructures with high churn such as hypotheses, thread-specific LIFO queues are used to recycle objects which are no longer used. This not only reduces memory wastage but re-uses recent objects which are likely to be in the CPU cache memory.

[3] http://boost.org/

[4] http://goog-perftools.sourceforge.net/doc/tcmalloc.html

Over 24% of the Moses decoder running time is spent on memory management and this increases to 39% when 32 threads are used, Table 4, dampening the scalability of the decoder. By contrast, our decoder spends 11% on memory management and does not significantly increase with more threads.

	Moses		Our Work	
# threads	1	32	1	32
Memory	24%	39%	11%	13%
LM	12%	2%	47%	38%
Phrase-table	9%	5%	2%	4%
Lex RO	8%	2%	2%	2%
Search	2%	0%	14%	19%
Misc/Unknown	45%	39%	24%	29%

Table 4: Profile of %age decoding time

Figure 2 compares the decoding time for Moses and our decoder, using the same models, parameters and test set. Our decoder is over 3 times faster with one thread, and 4.7 times faster using all cores. Like Moses, performance actually worsens after approximately 15 threads, however, the problem is not as pronounced. This gives us a better foundation on which to build further innovations for fast, multi-core decoding.

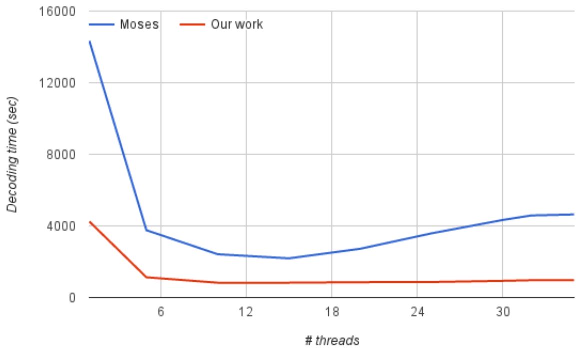

Figure 2: Decoding time of Moses and our decoder, using the same models

5.2 Stack Configuration

The most popular stack configuration for phrase-based model, as implemented in Pharaoh, Moses and Joshua, is coverage cardinality, ie. hypotheses that have translated the same number of source words are stored in the same stack. There have been research into other stack layouts such as (Ortiz-Martínez et al., 2006), and it has been noted that the decoder in (Brown et al., 1993) uses coverage stacks, as opposed to coverage cardinality.

We also note that the distortion limit which constrains hypothesis extension is dependent on the hypothesis' coverage vector, C *and* the end position of most recent source word that has been translated, e. The distortion limit must be checked for every instance of a hypothesis and translation rule, Figure 3. However, by separating hypotheses into set of hypotheses (*'ministacks'*) according to coverage and end position, the distortion limit only needs to be checked for each ministack, Figure 4. Furthermore, stack pruning is done on each of these hypotheses set therefore, changing how hypotheses are grouped can affect model scores. We therefore looked at the effects of three stack configurations:

```
for all $hypo$ in $stack_{|C|}$ do
  for all translation rules do
    if can-expand($C(hypo)$, $e(hypo)$, translation rule range) then
      expand hypo with translation rule → new hypo
      add new hypo to next stack
    end if
  end for
end for
```

Figure 3: Hypothesis Expansion with Cardinality Stacks

```
for all $ministack_{C,e}$ in $stack_{|C|}$ do
  for all translation rules do
    if can-expand($C$, $e$, translation rule range) then
      for all $hypo$ in $ministack_{C,e}$ do
        expand hypo with translation rule → new hypo
        add new hypo to next ministack
      end for
    end if
  end for
end for
```

Figure 4: Hypothesis Expansion with Coverage & End Position Stacks

1. coverage cardinality,
2. coverage,
3. coverage and end position of most recent translated source word.

Table 5 and Figure 5 present the tradeoff between decoding time and average model at various pop-limits.

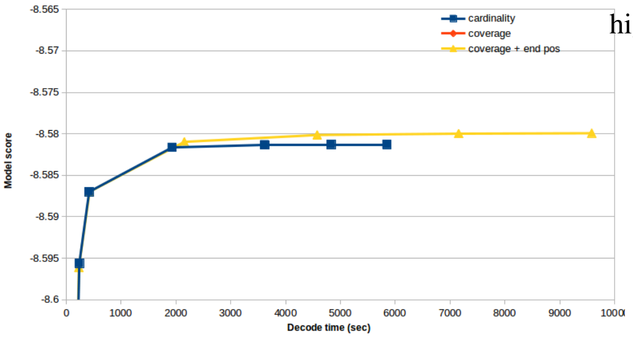

Figure 5: Trade-off between decoding time average model scores for different stack configurations

As can be seen, the model scores for all stack configurations are identical for low pop-limits parameters but grouping hypotheses into coverage & end position produces higher model scores for higher pop-limits. It is also slower but the time/quality tradeoff is better overall with this stack configuration. For lower pop-limits this configuration is slightly slower but not by much.

5.3 Translation Model

The Moses translation model caches the most recently queried translation rules for later re-use. This has been shown to perform badly for fast phrase-tables such as those described by (Bogoychev, 2013). We explore a simple caching mechanism that populates the cache during loading with rules that translates the most common source phrases. The static cache does not require the overhead of managing the most recently queried lookups but there is still some overhead in using a cache. Overall however, there was over a 10% decrease in decoding time using the optimum cache size, Table 6.

In the second optimatization, we disable the compression. This increase the size of the binary files from 17GB to 23GB but the time saved not needing to decompress the data resulted in a 1.5% decrease in decoding time with 1 thread and nearly 7% when the CPUs are saturated, Table 7.

5.4 Lexicalized Reordering Model

The lexicalized reordering model assign a probability to a translation rule, given the relative ordering of the rule in the hypothesis.

We integrate the lexicalized model file into the translation model but storing the model's probabilities in the phrase-table. This resulted in a significant decrease in decoding time, especially with high number of cores, Figure 6. Integrating the lexicalized reordering model into the translation model decreases decoding time by 29% with a single core but it is over 5 times faster using all cores. In fact, the decoding time with the integrated model is similar to that *without* a lexilized reordering model. Critically for systems with multicores servers, it enables the decoder to continue to scale, making efficient use of all available cores. This is in contrast to using a separate lexicalized reordering model where decoding time flatten out and actually worsens after approximately 15 threads.

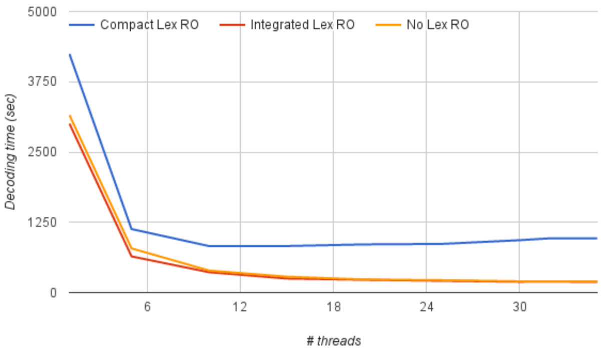

Figure 6: Decoding time with Compact Lexicalized Reordering, and integrated into a model the phrase-table

5.5 Scalability

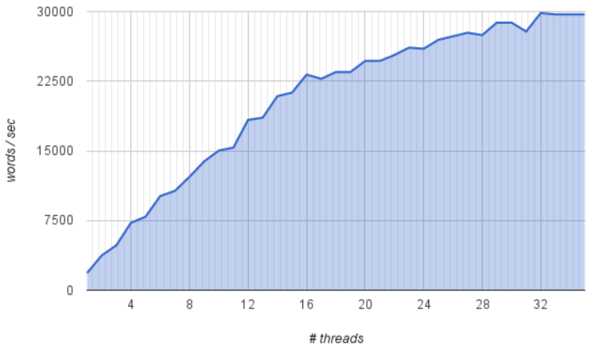

Figure 7: Our decoder's decoding speed

Figure 7 shows decoding speed against the number of threads, measured in words translated

Pop-limit	Cardinality		Coverage		Coverage & end pos	
	Time	Score	Time	Score	Time	Score
100	73	-8.64513	75	-8.64513	72	-8.64513
500	237	-8.59563	225	-8.59563	229	-8.59612
1,000	416	-8.58700	397	-8.58700	423	-8.58700
5,000	1930	-8.58165	1931	-8.58165	2153	-8.58098
10,000	3619	-8.58133	3630	-8.58133	4576	-8.58015
15,000	4830	-8.58130	5001	-8.58130	7156	-8.57999
20,000	5849	-8.58130	5916	-8.58130	9583	-8.57994

Table 5: Decoding time (in secs with 32 threads) and average model scores for different stack configurations

Cache size	Decoding Time	Cache Hit %age
Before caching	229	N/A
0	239 (+4.4%)	0%
1,000	213 (-7.0%)	11%
2,000	204 (-10.9%)	13%
4,000	205 (-10.5%)	14%
10,000	207 (-9.7%)	17%

Table 6: Decoding time (in secs with 32 threads) for varying cache sizes

# threads	Compressed pt	Non-compressed pt
1	3052	3006 (-1.5%)
5	756	644 (-14.8%)
10	372	362 (-2.7%)
15	284	250 (-12.0%)
20	244	227 (-7.0%)
25	218	209 (-4.1%)
30	206	192 (-6.8%)
35	203	189 (-6.9%)

Table 7: Decoding time (in secs with 32 threads) for compressed and non-compressed phrase-tables

per second. There is a constant increase in decoding speed when mrore threads are used, only decreasing slightly after 16 threads when hyperthreads are used. Overall, decoding is 12.5 times faster than single-threaded decoding when all 16 cores (32 hyperthreads) are fully utilized.

Overall though, the scalability is remarkably good, the decoder is able to make full use of all real and virtual cores. When all 16 cores are saturated with 2 hyper-thread each, the decoder is 16 times faster than single-threaded decoding. It is also 4.5 times faster than Moses with a single-thread and 9.6 faster when all cores are used.

This contrast with Moses where speed increases to approximately 16 threads but then actually become slower thereafter, Figure 8.

6 Other Models and Even More Cores

Our decoder show no scalability issues when we tested with the same model and tested set on a larger server, Figure 9.

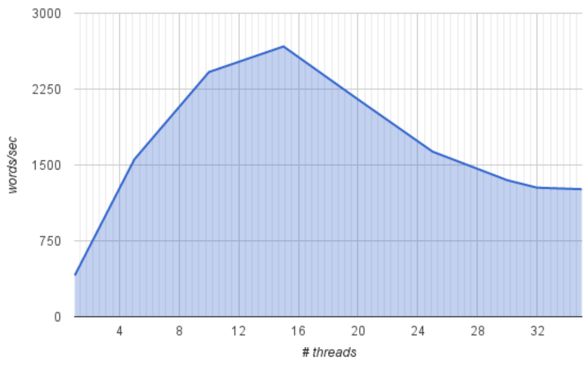

Figure 8: Moses' decoding speed

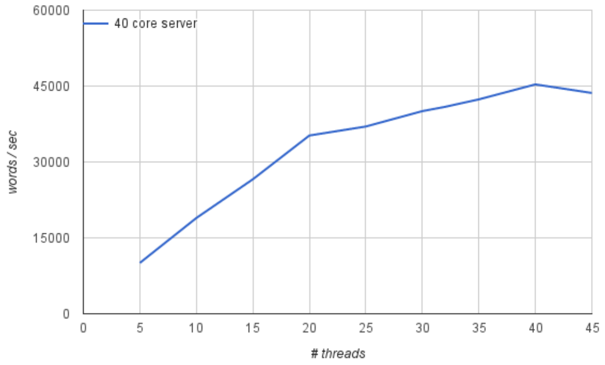

Figure 9: Decoding speed for with bigger servers

We verify the results with the French-English phrase-based system and test set. The speed gains are even greater than the Arabic-English test scenario, Figure 10. Our decoder is 5.4 times faster than Moses with a single-thread and 14.5 faster when all cores are saturated.

7 Conclusion

We have presented a new decoder that is compatible with Moses. By studying the shortcomings of the current implementation, we are able to op-

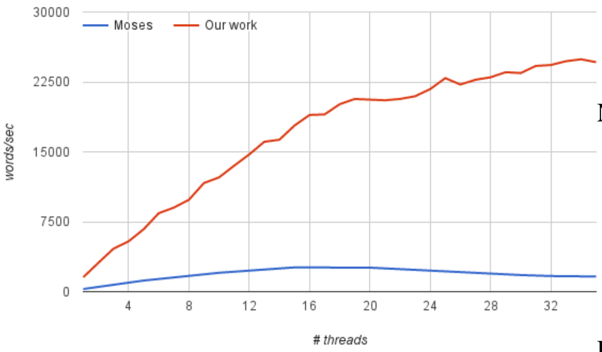

Figure 10: Decoding speed for fr-en model

timize for speed, particularly for multicore operation. This resulted in double digit gains compared to Moses on the same hardware. Our implementation is also unaffected by scalability issues that has afflicted Moses.

In future, we shall investigate other major components of the decoding algorithm, particularly the language model which has not been touched in this paper. We shall also explore the underlying reasons for the scalability issues in Moses to get a better understanding where potential performance issues can arise. This has application to other algorithms beside MT decoding.

References

Nikolay Bogoychev. 2013. Big data: Fast, small, and read only lookup. University of Edinburgh. Undergraduate Thesis.

Peter F. Brown, Stephen A. Della-Pietra, Vincent J. Della-Pietra, and Robert L. Mercer. 1993. The mathematics of statistical machine translation. *Computational Linguistics*, 19(2):263–313.

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

M. Fernández, Juan C. Pichel, José C. Cabaleiro, and Tomás F. Pena. 2016. Boosting performance of a statistical machine translation system using dynamic parallelism. *Journal of Computational Science*, 13:37–48.

Kenneth Heafield, Michael Kayser, and Christopher D. Manning. 2014. Faster Phrase-Based decoding by refining feature state. In *Proceedings of the Association for Computational Linguistics*, Baltimore, MD, USA, June.

Kenneth Heafield. 2011. KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, United Kingdom, July.

Marcin Junczys-Dowmunt. 2012. A space-efficient phrase table implementation using minimal perfect hash functions. In Petr Sojka, Ales Hork, Ivan Kopecek, and Karel Pala, editors, *15th International Conference on Text, Speech and Dialogue (TSD)*, volume 7499 of *Lecture Notes in Computer Science*, pages 320–327. Springer.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic, June. Association for Computational Linguistics.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. 2009. Joshua: An open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Athens, Greece, March. Association for Computational Linguistics.

Daniel Ortiz-Martínez, Ismael García-Varea, and Francisco Casacuberta. 2006. Generalized stack decoding algorithms for statistical machine translation. In *Proceedings on the Workshop on Statistical Machine Translation*, pages 64–71, New York City, June. Association for Computational Linguistics.

Joern Wuebker Matthias Huck Stephan Peitz, Malte Nuhn, Markus Freitag Jan-Thorsten Peter Saab, and Mansour Hermann Ney. 2012. Jane 2: Open source phrase-based and hierarchical statistical machine translation. In *24th International Conference on Computational Linguistics*, page 483. Citeseer.

Daniel Cer Spence Green and Christopher D Manning. 2014. Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 114–121. Citeseer.

Jörg Tiedemann. 2012. Parallel data, tools and interfaces in opus. In *LREC*, pages 2214–2218.

Joern Wuebker, Hermann Ney, and Richard Zens. 2012. Fast and scalable decoding with language model look-ahead for phrase-based statistical machine translation. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2*, pages 28–32. Association for Computational Linguistics.

Makoto Yasuhara, Toru Tanaka, Jun-ya Norimatsu, and Mikio Yamamoto. 2013. An efficient language model using double-array structures. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 222–232, Seattle, Washington, USA, October. Association for Computational Linguistics.

Richard Zens and Hermann Ney. 2007. Efficient phrase-table representation for machine translation with applications to online mt and speech translation. In *HLT-NAACL*, pages 492–499.