Fast and highly parallelizable phrase table for statistical machine translation

Nikolay Bogoychev
The University of Edinburgh
11 Crichton St, Edinburgh EH8 9LE
United Kingdom
n.bogoych@ed.ac.uk

Hieu Hoang
Moses Machine Translation CIC
hieu@moses-mt.org

Abstract

Speed of access is a very important property for phrase tables in phrase based statistical machine translation as they are queried many times per sentence. In this paper we present a new standalone phrase table, optimized for query speed and memory locality. The phrase table is cache free and can optionally incorporate a reordering table within. We are able to achieve two times faster decoding by using our phrase table in the Moses decoder in place of the current state-of-the-art phrase table solution without sacrificing translation quality. Using a new, experimental version of Moses we are able to achieve 10 times faster decoding using our novel phrase table.

1 Introduction

Phrase tables are the most basic component of a statistical machine translation decoder, containing the parallel phrases necessary to perform phrase-based machine translation. Due to the noisy nature of phrase extraction and the large phrase vocabulary, phrase tables' size can reach hundreds of gigabytes in size. Lopez (2008) describes phrase tables of size of half of terabyte. A decade ago it was prohibitively expensive for a phrase table of this size to reside in memory, even if hardware supported it: a gigabyte of RAM back in 2006 costed about a 100 USD, compared to 5 USD in 2016. Because of that for a long time Machine Translation was considered a big data problem and the engineering efforts were focused on reducing the model size. This lead to the creation of several binary phrase table implementations that tackled the memory usage problem: Zens and Ney (2007) and Junczys-Dowmunt (2012b) developed memory mapped phrase tables which also reduce memory usage using specific datastructures. The former uses a trie (Fredkin, 1960) and the latter uses specific for the purpose phrasal rank encoding. Lopez (2007) and Germann (2015) developed suffix array based phrase tables, which work directly with the parallel corpora in order to enable easier addition of new data, avoid long binarization times and keep memory usage low, but traditional precomputed phrase tables offer better performance. RAM prices have dropped 20 times over the past 10 years and high performance server machines have hundreds of gigabytes of memory. For those machines it is no longer needed to sacrifice query performance in favour of compression techniques such as the one in Junczys-Dowmunt (2012a). Furthermore the machines nowadays are highly parallel and locking caches which didn't hurt performance in the past now prevent implementations from scaling. We have designed a new phrase table called **ProbingPT** based on linear probing hash (Heafield, 2011) for storage and lock-free querying, in order to deliver the best possible performance in modern use cases where memory is not an issue.

2 Implementation

First, we will give a brief overview of Junczys-Dowmunt's (2012b) CompactPT which is currently the state of the art phrase table in terms of both speed and space usage. It uses phrasal rank compression (Junczys-Dowmunt, 2012a) which can be viewed as a form of byte pair encoding (Gage, 1994). The method recursively encodes bigger strings as a composition of several smaller ones until only small units remain. Minimum perfect hashing (Nick Cercone, 1983) is used to hash phrases to their expansions and on top of that bit aligned Huffman encoding is used to further compress the phrases. This approach achieves the smallest model size but it has several drawbacks when it comes to lookup. First, minimum perfect hashing requires a secondary hash function called fingerprinting in order to avoid false positives which results in increased CPU usage. Second, while phrasal rank encoding is extremely space efficient, it is quite slow to compute, because of the multitude of random memory accesses necessary to reconstruct a single phrase. The reason is that when a request to read a portion of memory is submitted what is actually fetched is not only the bytes that were requested but also the surrounding bytes. This is because usually when one byte of memory is accessed, the surrounding bytes would also be necessary so memory has been designed to fetch things in small burst, called DRAM bursts. As such peak memory performance can only be achieved by accessing consecutive memory and random memory accesses reduce the total memory bandwidth, because some of fetched bytes are not used.

In order to speed up querying in CompactPT, extensive caching is used but it is not thread local and causes a lot of locking for higher thread count. In our experiments we found that more than 8 threads actually hurt CompactPT's performance. The phrase table also has a mode which disables phrasal rank encoding and caching. In this mode performance at higher thread count doesn't decrease but instead flattens out, however it is unable to achieve better performance than the phrase rank encoding version no matter the thread count.[1] Even if caches don't cause lock contention at higher thread count, they carry additional overhead during runtime. Our goal in design was to eliminate the necessity for cache by using high performance datastructures and eliminate random memory accesses to maximize the memory bandwidth.

2.1 ProbingPT

Our phrase table is based on an existing linear probing hash table implementation (Heafield, 2011). Linear probing hash provides $O(1)$ search time, has a very small overhead per entry stored and is shown to be very fast in practice (Heafield, 2011). The phrase table consists of two byte arrays: The first contains the probing hash table and the second one contains the payloads (phrase probabilities, word alignments and optionally lexically reordering scores) associated with each entry in the hash table. Hashes of the source phrases are used as keys. When the phrase table is queried, the source phrase is hashed and we try to find it in the probing hash table. If it is found inside the hash table we are given a start and end index corresponding to the location of the target phrases associated with the source phrase queried inside the payloads byte array. The payloads byte array stores consecutively in binary format each target phrase together with its scores and word alignment information. We have also provided the option to store lexical reordering information and sparse features. Unlike previous phrase tables implementations, this phrase table doesn't employ any compression method which allows for all target phrases associated with a single source phrase to be fetched in a single memory operation. In contrast, both Junczys-Dowmunt's (2012b) and Zens and Ney (2007) employ pointer chasing during querying in order to extract and reassemble the results. Their approaches are more space-efficient but incur higher memory cost due to increased number of random memory accesses. Furthermore our implementation doesn't require any scratch memory to decompress queries: they can be read directly from the payloads byte array which contributes to its speed and avoids extra memory operations (allocations/deallocations) or the need for caching. Storing lexical reordering information inside the phrase table reduces the memory usage, because we no longer need to store a key for every lexical reordering score, as we reuse the phrase table key. Extracting lexical reordering scores no longer carries an extra performance penalty as querying is tied to the phrase table query and all related scores would be fetched with the same DRAM burst, because they are stored consecutively in memory. To our knowledge, this is the first phrase table implementation that incorporates lexical reordering table.

The phrase table is part of upstream Moses[2] but it can also be used standalone.[3]

3 Experimental setup

For our performance evaluation we used French-English model trained on 2 million EUROPARL sentences. We used a KenLM (Heafield, 2011)

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

[2]Anonymous for submission

[3]Anonymous for submission

language model and cube pruning algorithm (Chiang, 2007) with a pop-limit of 400. We time the end to end translation of 200,000 sentences from the training set. All experiments were performed on a machine with two Xeon E5-2680 processors clocked at 2.7 Ghz with total of 16 cores and 16 hyperthreads and 290 GB of RAM. In all of our figures "32 cores" means 16 cores and 16 hyperthreads. Note that hyperthread do not provide additional computational power but merely permit better resource utilization by allowing more work to be scheduled for the CPU by the OS. This allows the CPU to already have scheduled work to do while a scheduled process is waiting for IO. Using hyperthreads will not necessarily increase performance and in cases with high lock contention it can be detrimental for performance.

3.1 Decoders

We use two different decoders for our experiments: The widely used moses machine translation decoder, available publically and *Moses2*, an experimental faster version of Moses.[4] We perform some benchmarks using Moses to show the speedup our implementation offers as a drop-in replacement to existing phrase tables in the widely used decoder. Unfortunately Moses has known multi-threading issues that come from the usage of several functions which call **std::locale** as part of their initiations, which carries a global lock.[1] As such it is not entirely adequate to use it to measure the performance of the phrase tables because it serves as a bottleneck that might hide performance issues. Thus we used the highly optimized *Moses2* to show the speed our phrase table can achieve when it is running on a fast decoder, optimized for multi-threading. Furthermore because of their intrusive nature, integrated lexical reordering tables are only implemented in *Moses2*. It is expected that when *Moses2* matures it will be merged back into Moses master.

3.2 PhraseTables

In our experiments we focus on comparing ProbingPT against CompactPT. There are currently two other phrase tables: PhraseDictionaryOnDisk, a multithreading enabled implementation of the Zens and Ney (2007) phrase table and PhraseDictionaryMemory, an in-memory phrase table which reads in the raw phrase table and puts it inside a hash map. Junczys-Dowmunt (2012b) has shown that CompactPT is faster than PhraseDictionaryOnDisk under any condition, so we do not run experiments against it. PhraseDictionaryMemory comes with the downside that it needs to parse in the phrase table first, before decoding can commence, which leads to long loading times and huge memory usage. In our experimental setup, the in-memory phrase table took 20 minutes to load and consumed 86 GBs of memory, more than ten times more memory than any other phrase table. Even when disregarding loading time, we found out that it is consistently 1-5% slower than ProbingPT in various thread configurations. We decided not to include those results, as they do not show anything interesting and because of the aforementioned shortcomings, PhraseDictionaryMemory is seldom used in practice, unless the dataset involved is really tiny.

ProbingPT and CompactPT produced identical translations under the same decoder. In our tests 3 out of 200,000 sentences slightly differ in their translation. This is expected according to Junczys-Dowmunt (2012b) because CompactPT's fingerprinting leads to collisions and extracting the wrong phrase in few rare cases. We conclude that our implementation is correct and can be used as drop-in replacement for CompactPT. We have provided the complete set of conducted experiments on Figure 5 in the appendix. Those are useful if the reader wishes to compare system/user time usage between different configurations.

3.3 Model sizes

Phrase table	Size
ProbingPT	5.8 GB
ProbingPT + Reordering (RO)	8.2 GB
CompactPT	1.3 GB
CompactPT RO	0.6 GB

Table 1: Phrase table sizes

CompactPT which is designed to minimize model size has naturally lower model size compared to ProbingPT. However the extra RAM used is only 2% of the available on our test system which is insignificant. Using the extra memory is justified by the increased performance.

[4]Anonymous for submission

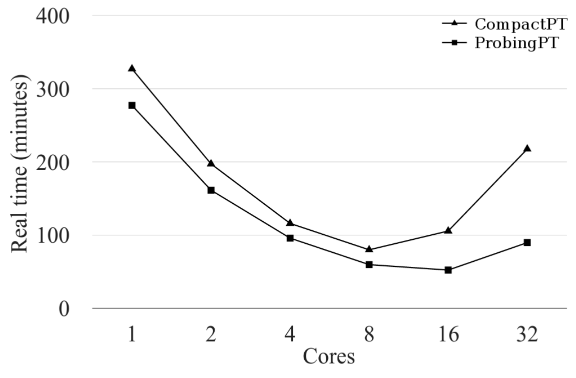

Figure 1: Real time comparison of Moses between ProbingPT and CompactPT together with reordering models based on CompactPT.

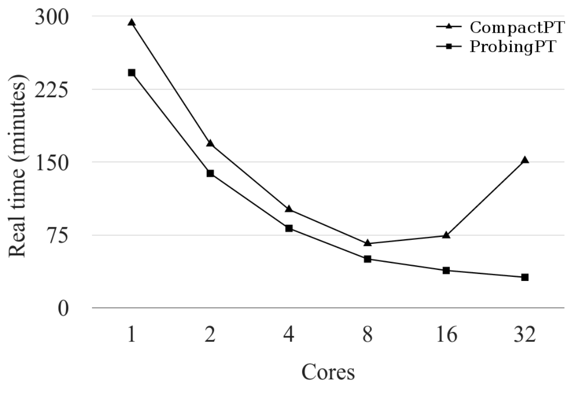

Figure 2: Real time comparison of Moses between ProbingPT and CompactPT

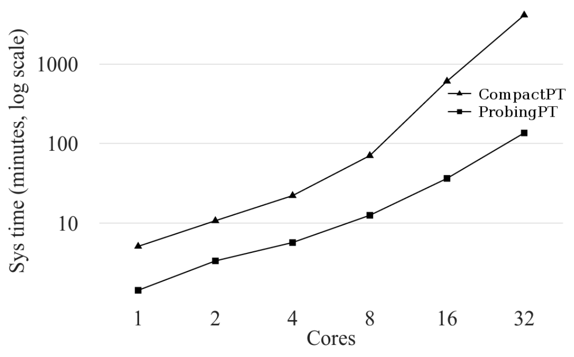

Figure 3: System time comparison of the systems on Figure 2. The comparison is in log scale.

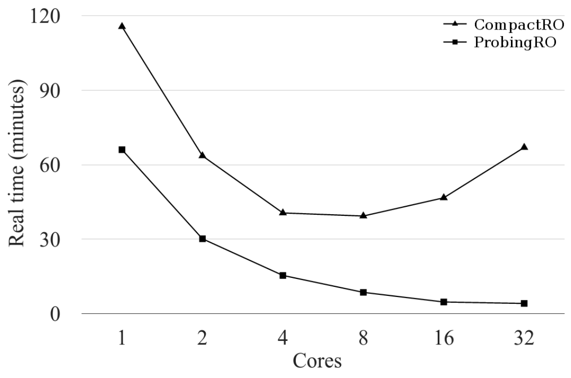

Figure 4: *Moses2* comparison between ProbingPT integrated reordering and CompactPT based reordering. Both systems use ProbingPT as a phrase table.

4 Evaluation

Figure 1 shows performance comparison of two systems with CompactPT based reordering tables that differ in the phrase table used. The best performing ProbingPT system here delivers about 30% better performance compared to the corresponding CompactPT system. We see that the CompactPT system doesn't improve its performance when using more than 8 threads, but the ProbingPT one continues to scale further until it starts using hyperthreads.

We find it likely that the performance of the ProbingPT system on Figure 1 is hampered by the inclusion of CompactPT based reordering. Moses doesn't support ProbingPT based reordering and in order to measure the head-to-head performance of the two phrase tables we conducted the same test using two systems that do not use reordering tables and only differ by the phrase table, as shown on Figure 2. We can see that ProbingPT consistently outperforms CompactPT by 10-20% at lower thread count but the difference grows as much as 5 times in favour of ProbingPT at the maximum available thread count on the system. If we compare the best performance achieved from both system, ProbingPT is capable of delivering twice the performance of CompactPT. It is important to note that ProbingPT's performance always increases with the increase of the thread count, whereas CompactPT's performance doesn't improve past 8 threads. We can also see that the ProbingPT based system can even take advantage of hyperthreads, which is not possible with any system that uses CompactPT based table (Figure 1). On Table 2 we can observe that removing the reordering table from the CompactPT system has a much smaller effect than removing it from the ProbingPT system. This hints that lexicalized reordering only slows down the decoder because it is implemented in a inefficient manner. We can conclude that Moses can achieve faster translation times on highly parallel systems by using ProbingPT.

4.1 Why is CompactPT slower?

In the single-threaded case it likely that CompactPT's many random memory accesses cause it to be slower than ProbingPT, because consecutive memory accesses are much faster due to the DRAM burst effect. When the thread count grows, the performance gap between CompactPT and ProbingPT widens, because of the locking that goes on in the former's cache. This can be seen from Figure 3 which shows the system time used during the execution of the phrase table only test. System time shows how much time a process has spent inside kernel routines, which includes but is not limited to locking and memory allocation. The ProbingPT system uses orders of magnitude less system time compared to the CompactPT one. The system time used in the CompactPT system grows linearly until 8 threads and then the growth rate starts increasing at a faster rate widening the gap with ProbingPT. This is also the reason why CompactPT's performance severely degrades when using hyperthreads. The ProbingPT system on the other hand (Figure 3) increases its usage of system time at a linear rate even when using hyperthreads. We can conclude that the simpler design of ProbingPT scales very well with the increase of number of threads and is suitable for use in modern translation systems running on contemporary hardware.

4.2 Integrated reordering table

As integrated lexical reordering is only available in *Moses2* we conducted an experiment where we compare systems using CompactPT based reordering and ProbingPT integrated reordering (Figure 4). The best ProbingPT based system is able to translate all sentences in our test set in only 4 minutes, whereas the best CompactPT reordering system took 39 minutes (Table 3). We also observed limited scaling when using CompactPT based reordering: the best performance was achieved at 8 threads. For contrast with Moses (Table 2) we can see that lexicalized reordering has negligeble impact on performance if it is used within ProbingPT (We believe the reason we are getting slightly worse results when not using a reordering table are due to a bug in our implementation). We are not entirely certain which factor contributed more to the increased performance: having a reordering table based on the faster ProbingPT or the reduced IO and computational resources that the integrated reordering table requires. As we do not currently have a standalone ProbingPT based reordering table we can not say for sure. Nevertheless we achieve 10x speedup by using our novel reordering table within *Moses2*.

4.3 Profiling the code

We were very surprised of the speedup our phrase table offered, particularly in *Moses2*, because in phrase based decoding, the number of phrase table queries increases linearly with the length of the sentence. They constitute a tiny fraction of the number of language model queries, which are about 1 million per sentence (Heafield, 2013). We decided to investigate our results using Google's profiler.[5] We profiled the pair of systems, displayed on Figure 4, because they showed the highest relative difference between each other. In the system which has ProbingPT based reordering, the language model is responsible for about 40% of the decoding runtime, compared with only 1% in the *Moses2* system with CompactPT based reordering. In the latter system the runtime is dominated by CompactPT search and **std::locale** locking due to the phrase table using string operations during its search.

In Moses the difference between using ProbingPT and CompactPT is not so apparent, before we go to higher thread count, because the decoder itself is very slow and hides the phrase table inefficiencies. It is clear that even though the phrase table queries are a small part of the full decoding process, they are enough to slow it down 10 times if no other bottlenecks exist. Using ProbingPT for both the phrase table and the reordering model makes for a compelling combination.

5 Future work

In the future we will add support for hierarchical phrase tables inside ProbingPT. In hierarchical machine translation the burden on the phrase table is a lot higher so the improved performance would be even more noticeable. Given the difference between the systems with and without ProbingPT based reordering in Table 3 we believe that adding that feature to Moses will allow us to get performance similar to that in the final column of

[5]https://github.com/gperftools/gperftools

Cores	CompactPT, RO	CompactPT, NoRO	ProbingPT, RO	ProbingPT, NoRO
1	327	300	277	242
2	197	167	161	138
4	116	101	96	82
8	80	66	60	50
16	106	74	52	39
32	218	151	90	31

Table 2: Time (in minutes) it took to translate our test set with Moses with different number of cores used. The systems differ by the type of phrase table used (ProbingPT or CompactPT) and whether they use a reordering table (based on CompactPT). The fastest translation time for each system is highligthed.

Cores	CompactRO	ProbingRO	NoRO
1	116	66	72
2	64	30	35
4	40	15	18
8	39	9	10
16	46	5	6
32	67	4	5

Table 3: Time (in minutes) it took to translate our test set with *Moses2* with different number of cores used. Since the only phrase table that is used is ProbingPT, the systems differ by the reordering table used. The fastest translation time for each system is highligthed.

Table 2, while maintaining the quality of the more complex model described in the first column of the same table.

6 Conclusion

As hardware evolves extremely fast, it may prove useful to revisit old problems which are considered solved. The new available technology compels us to reconsider our priorities and decisions we took in the past.

We designed a faster phrase table that is able to take full advantage of the modern highly parallel CPUs. It shows better performance than related work and also scales better with higher thread count and it helped us expose performance issues in Moses. We believe ProbingPT is useful to industry and researchers who use modern server machines with many cores and a lot of main memory. Enthusiast machine translation users would probably prefer to use CompactPT as it is most suitable when memory is limited and the thread count is low.

Acknowledgements

This work was conducted within the scope of the Horizon 2020 Innovation Action *Modern MT*, which has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 645487.

This work is sponsored by the Air Force Research Laboratory, prime contract FA8650-11-C-6160. The views and conclusions contained in this document are those of the authors and should not be interpreted as representative of the official policies, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

We thank Adam Lopez, Kenneth Heafield, Ulrich Germann, Rico Sennrich, Marcin Junczys-Dowmunt, Nathan Schneider, Sorcha Gilroy, Clara Vania and the anonymous reviewers for productive discussion of this work and helpful comments on previous drafts of the paper. Any errors are our own.

References

David Chiang. 2007. Hierarchical phrase-based translation. *Comput. Linguist.*, 33(2):201–228, June.

Edward Fredkin. 1960. Trie memory. *Commun. ACM*, 3(9):490–499, sep.

Philip Gage. 1994. A new algorithm for data compression. *C Users J.*, 12(2):23–38, February.

Ulrich Germann. 2015. Sampling phrase tables for the moses statistical machine translation system. *Prague Bull. Math. Linguistics*, 104:39–50.

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.

Kenneth Heafield. 2013. *Efficient Language Modeling Algorithms with Applications to Statistical Machine Translation*. Ph.D. thesis, Carnegie Mellon University, September.

Marcin Junczys-Dowmunt. 2012a. Phrasal rank-encoding: Exploiting phrase redundancy and translational relations for phrase table compression. *Prague Bull. Math. Linguistics*, 98:63–74.

Marcin Junczys-Dowmunt. 2012b. A space-efficient phrase table implementation using minimal perfect hash functions. In *Text, Speech and Dialogue - 15th International Conference, TSD 2012, Brno, Czech Republic, September 3-7, 2012. Proceedings*, pages 320–327.

Adam Lopez. 2007. Hierarchical phrase-based translation with suffix arrays. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 976–985, Prague, Czech Republic, June. Association for Computational Linguistics.

Adam Lopez, 2008. *Tera-Scale Translation Models via Pattern Matching*, pages 505–512. Coling 2008 Organizing Committee, 8.

John Boates Nick Cercone, Max Krause. 1983. Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *CAMWA*, 9(1):215–231.

Richard Zens and Hermann Ney. 2007. Efficient phrase-table representation for machine translation with applications to online mt and speech translation. In *Human Language Technology Conf. / North American Chapter of the Assoc. for Computational Linguistics Annual Meeting*, pages 492–499, Rochester, NY, April.

A Experiments Matrix

Figure 5: A matrix of all experiments conducted.

Cores/Variant	Moses1 + compactpt + compactreord	Moses1 + compactpt + no reord	moses1 + probing + compactreord	moses1 + probing + no reord	moses2 + probing + compactreord	moses2 + probing + integratedRO	moses2 + probing + no reord
	real 327m3.554s	real 292m44.73s	real 277m10.41s	real 241m51.16s	real 115m35.32s	real 66m2.933s	real 72m28.06s
	user 318m46.127s	user 287m36.13s	user 273m16.02s	user 240m23.27s	user 112m26.00s	user 63m18.006s	user 71m27.24s
1	sys 8m6.706s	sys 5m6.16s	sys 3m48.58s	sys 1m26.12s	sys 3m10.58s	sys 2m33.317s	sys 1m3.68s
	real 197m18.17s	real 168m36.05s	real 161m24.43s	real 138m8.04s	real 63m33.20s	real 30m9.780s	real 35m18.24s
	user 374m27.39s	user 325m28.96s	user 314m7.35s	user 272m17.22s	user 120m21.88s	user 59m27.849s	user 69m53.43s
2	sys 17m28.34s	sys 10m43.95s	sys 7m52.60s	sys 3m21.41s	sys 6m36.93s	sys 0m48.234s	sys 0m38.95s
	real 116m1.57s	real 101m4.74s	real 95m55.83s	real 81m43.00s	real 40m34.79s	real 15m24.667s	real 18m30.99s
	user 423m32.44s	user 378m36.98s	user 366m26.63s	user 319m20.53s	user 150m12.16s	user 61m5.165s	user 73m22.10s
4	sys 36m47.68s	sys 22m9.95s	sys 14m48.15s	sys 5m42.96s	sys 11m50.91s	sys 0m25.914s	sys 0m32.56s
	real 79m56.00s	real 66m0.99s	real 59m41.43s	real 50m14.49s	real 39m19.77s	real 8m35.280s	real 10m17.52s
	user 501m11.52s	user 449m3.61s	user 433m34.24s	user 383m49.20s	user 288m40.73s	user 67m57.790s	user 81m25.37s
8	sys 127m49.32s	sys 70m31.53s	sys 36m35.65s	sys 12m32.91s	sys 25m20.14s	sys 0m26.249s	sys 0m32.35s
	real 105m37.01s	real 74m10.77s	real 52m15.47s	real 38m25.06s	real 46m43.13s	real 4m42.100s	real 5m57.21s
	user 600m33.02s	user 562m59.64s	user 612m47.32s	user 557m28.61s	user 681m40.37s	user 74m3.060s	user 93m42.05s
16	sys 1072m12.89s	sys 607m53.22s	sys 205m19.05s	sys 36m28.92s	sys 64m17.28s	sys 0m30.157s	sys 0m42.36s
	real 217m58.10s	real 151m22.90s	real 89m46.79s	real 31m25.80s	real 66m54.80s	real 4m7.794s	real 5m12.37s
	user 779m59.18s	user 709m57.34s	user 668m15.93s	user 752m32.02s	user 233m23.18s	user 129m38.242s	user 163m29.34s
16 + 16 hyper	sys 6175m37.16s	sys 4115m3.50s	sys 2188m54.82s	sys 135m58.88s	sys 1903m43.26s	sys 0m57.126s	sys 1m21.11s