High Performance Computing

Francesco Benvenuto 858843
Marco Marangoni 858450
Mauro Molin 857855

January 2020

Contents

1 Introduction

Our goal for this project was to implement the algorithm presented in the RapidScorer paper, improving its parametrization by allowing custom bit sizes per document and SIMD[1] sizes.

Please note that to understand this document, understanding of the RapidScorer paper is required.

The repository with the code, the results and this PDF can be found here.

2 Model and testing

We chose to test our program on a decision forest with 10,000 decision trees, each with 400 leaves.

To create this model we used the LightGBM[2] library on the MSN Dataset[3] (136 features per document). Along with this data we also calculated the score for each document and saved it to a `test.txt` in order to allow the various algorithms to check the correctness of their solutions.

3 Algorithms

We developed several different algorithms, each time by improving the previous one trying to increase the overall performances.

3.1 Normal evaluation

The first implementation is a simple traversal of the decision tree. We did it for testing the parsing of the model and to better understand the problem at hand.

3.2 EqNodesRapidScorer

This is the first implementation based on the paper, but without any advanced optimization. In particular, we implemented the `EqNode` and `Epitome` classes described in the paper.

3.3 LinearizedRapidScorer

The first improvement we did was to scorporate the vector of `EqNodes` in four vectors of the fields containing `EqNode`. This is helpful because one of the steps of the algorithm is to find a feature threshold greater than the value of the document. Having all the thresholds in a AOS (array-of-struct) format prevents the compiler to vectorize the comparison.

Moreover, having all the threshold in a single array, allowed us to add another optimization by performing a binary search, thus reducing the number of comparisons and memory accesses from $O(n)$ to $O(log(n))$.

Unfortunately the overall complexity remains the same because it's bounded by the `Epitome_AND` function.

3.4 MergedRapidScorer

In this implementation we further improved the performance by implementing all the strategies presented in the paper. In particular we implemented:

- `Epitome_Short` by scorporating the vector of epitomes in four arrays, allowing us to store the `lb` and `lbp` iff they are different from `fb` and `fbp`; by doing so we avoided to store between 75% and 95% of `lb` and `lbp`, based on the configuration.
- Equivalent nodes merging: a similar reasoning can also be applied to the feature thresholds, allowing us to avoid storing duplicates values in the threshold array. In practice this reduced the number of `doubles` stored by $95\% - 99.5\%$.

[1] https://en.wikipedia.org/wiki/SIMD
[2] https://github.com/microsoft/LightGBM
[3] https://www.microsoft.com/en-us/research/project/mslr/

3.5 SIMD

The next (and last) improvement we made to the program is by using SIMD explicitly, enabling multiple documents (up to 64) to be evaluated at the same time. For this program we used the `AVX512`[4] Intel instruction set. This implementation is based on the MergedRapidScorer, but without the binary search, because of two reasons:

- It wouldn't actually be beneficial: since the number of thresholds is around 250, and performing one linear search is better than 64 binary searches ($250 < 64 * log_2(250)$);
- With the linear search we can directly use the instruction `VCMPPD` that creates the mask used to perform the `Epitome_AND` directly.

4 Parallelization strategies

In this section we will describe the three different parallelization strategies we tried.

4.1 Parallel features

This is the most straightforward (and slow) way to parallelize all the algorithms. Instead of computing the `for` on each feature sequentially (line 4 of algorithm 3 in the paper), we used OpenMP[5] to parallelize it.

This implementation requires to manage explicitly the critical zone for updating the `leafindex` bitmask that is shared among all the threads. Since OpenMP doesn't support atomic updates on SIMD registers, we couldn't implement this strategy for the SIMD algorithm. We decided to not introduce any other locking mechanism for SIMD, since the performance of this strategy is not very good anyway.

Furthermore, when parallelizing by feature, summing the scores of each tree is also parallelized.

4.2 Parallel documents

This strategy consists in evaluating more documents at the same time, using OpenMP. This doesn't require any synchronization since the memory among the threads is read-only, and each thread has its own `leafindex`.

For obvious reasons, while using this strategy more data must be kept in memory at the same time, since there are multiple `leafindex`es.

4.3 Parallel forests

This strategy consists in splitting the decision forest in multiple smaller forests that will be evaluated at the same time. This implementation has a drawback with respect to the previous one: we introduced a synchronization barrier right before computing the score of the document because we need to wait for all the partial scores on each smaller forest to complete. On the other hand, this strategy doesn't require additional memory with respect to the sequential implementation. These two differences will cause the performances sometimes to be better on parallel forests and sometime to be better on parallel documents.

5 Parametrization

While in the paper implementation the size of the bitmasks (e.g. epitome) was fixed to 8 bits, in our implementation it's parametrized and allows the caller to choose between 8, 16, 32 or 64 bits. With this flexibility we've shown evidence that it exists a performance boost by choosing bigger sizes.

Similarly, we parametrized the size of the SIMD register to use, which can be 128, 256 or 512 bits long.

All the parameterizable options can be configured by creating and passing an instance of the `Config` class. While doing so we made sure not to introduce any performance issue (for example, we didn't use virtual functions).

[4] https://software.intel.com/en-us/articles/intel-avx-512-instructions
[5] https://www.openmp.org/

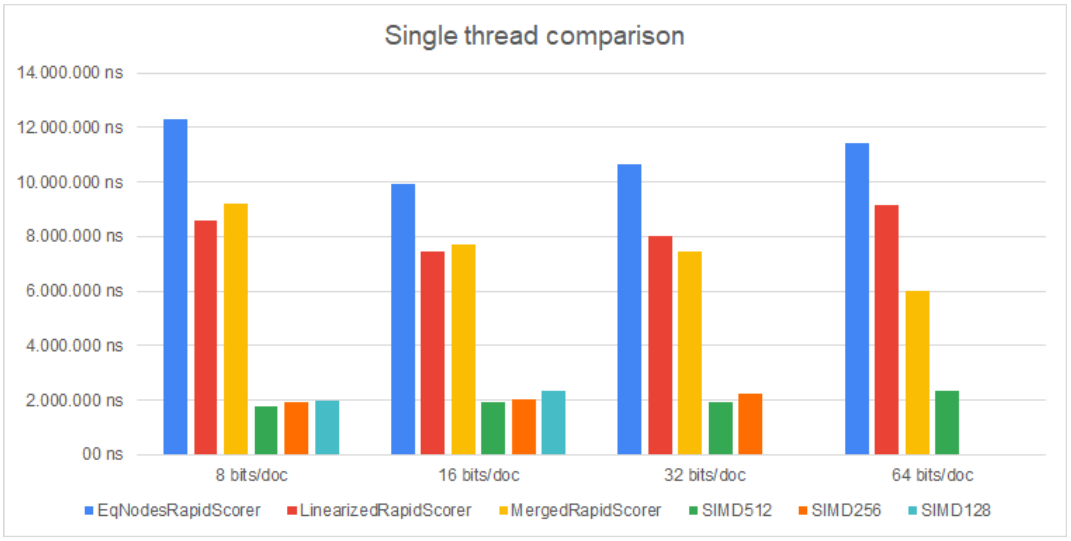
6 Results

In this section we will discuss the performance results of some relevant configurations.

All the tests were performed on an AWS machine of type `c5.9xlarge` with the following configuration:

- 36 vCPUs of Intel Xeon Platinum 8000 series with about 33 MiB of L3 cache
- 72 GiB of RAM
- SSD storage

6.1 Single thread comparison



Graph 1: Single thread comparison

In Graph 1 we compare all implementations using only a single thread, grouped by how many bits are used by `fb` and `lb` of the epitome. It's immediately noticeable how the SIMD implementation always outperforms the others.

The `EqNodesRapidScorer` algorithm is always the slowest. This is because the memory layout is not optimal.

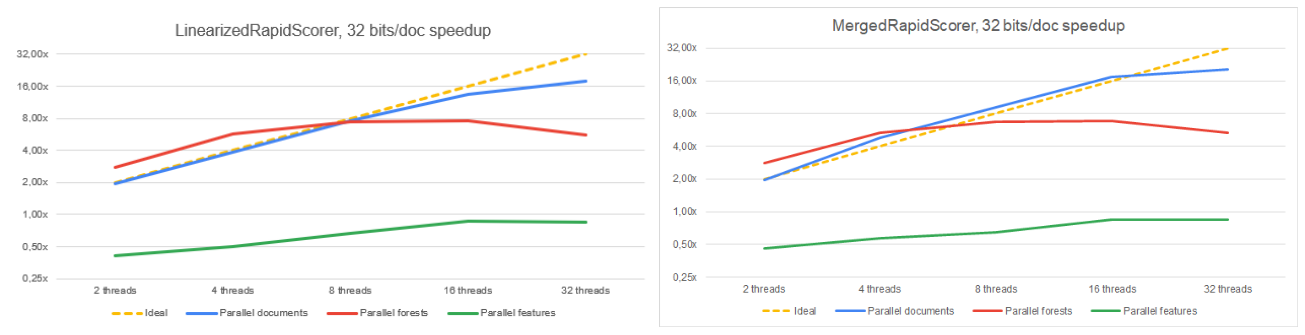
The `LinearizedRapidScorer` and `MergedRapidScorer` are always comparable in speed, except in the last group. This is because the smaller memory footprint of the latter gives a significant boost in performance.

In SIMD, we can notice how it's always faster to use less bits per document, as we can then compute more documents in parallel.

As already mentioned before, when using `MergedRapidScorer`, assigning more bits for the epitome increases the overall performances. This is because `MergedRapidScorer` uses few memory, so the penalty of needing to store more data is less than the improvement of computing more bits of the `leafindex` at the same time.

6.2 Scalability speedups

6.2.1 `LinearizedRapidScorer` and `MergedRapidScorer`



Graph 2: 32 bits/doc speedup of `LinearizedRapidScorer` and `MergedRapidScorer`

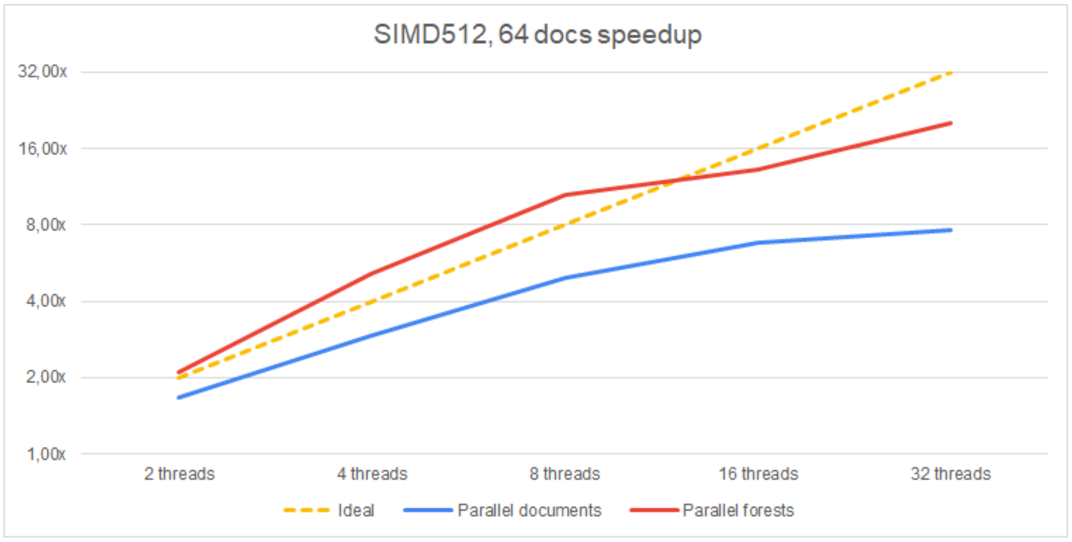
For both algorithms the parallel feature strategy is always slower then the single thread one as shown in Graph 2. This is because the overhead associated with the atomic operation is greater than the gain. This is justified from the fact that the computation is bounded by the `Epitome_AND` function.

After 4 threads the performance of the parallel forests strategy starts to deteriorate because the synchronization barrier involves more and more threads.

The parallel documents strategy has a linear speedup up to 16 threads, (*super-linear* for `MergedRapidScorer`). After that, the number of documents that must be stored in memory increases so much that the benefit of adding more threads starts to decrease. For the `MergedRapidScorer` the speedup is greater because of the smaller memory footprint.

The same considerations also apply to different bits/doc sizes.

6.2.2 `SIMD`



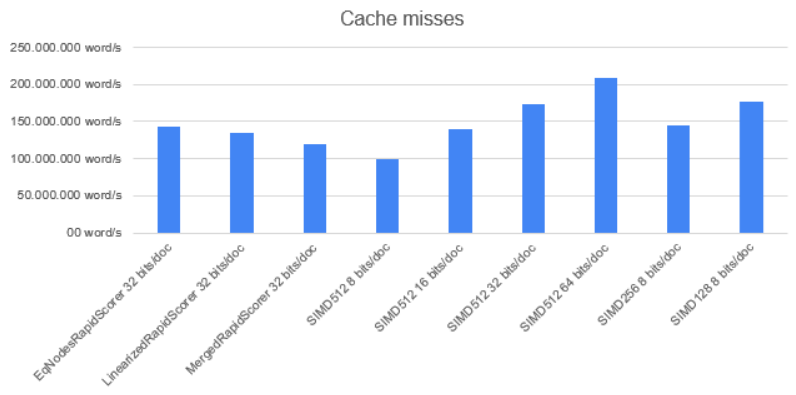
Graph 3: `SIMD512` 8 bits/doc speedup

As said before we didn't implement the parallel features strategy with SIMD.

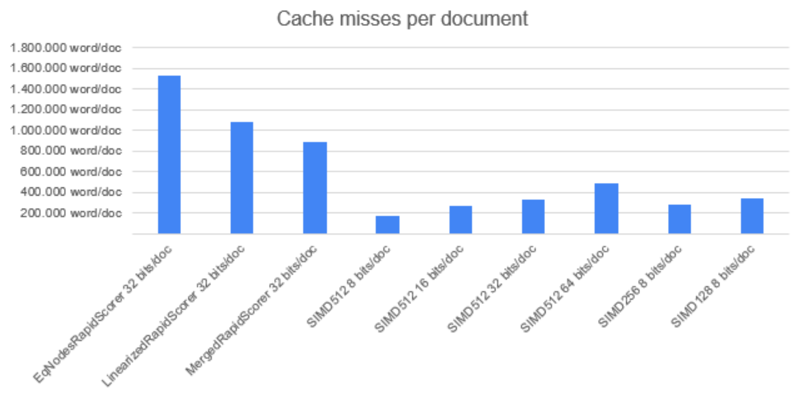
Unlike the previous algorithms, the speedup for parallel forests is always greater than the one of parallel documents, as we can see in Graph 3. This is justifiable by the fact that to compute more documents at the same time a lot of memory is required. For example, for 32 threads in the parallel documents setup we require $64\ docs * 32\ threads * 400\ leaves * 10000\ trees = 976\ MiB$ to store the leaf indexes. On the other hand parallelizing on the forests (all the 32 threads work on the same 64 documents) will require only $\frac{976\ MiB}{32} = 30\ MiB$ of memory.

6.3 Cache analysis

6.3.1 Algorithm comparison



Graph 4: Cache word misses per second of different algorithms



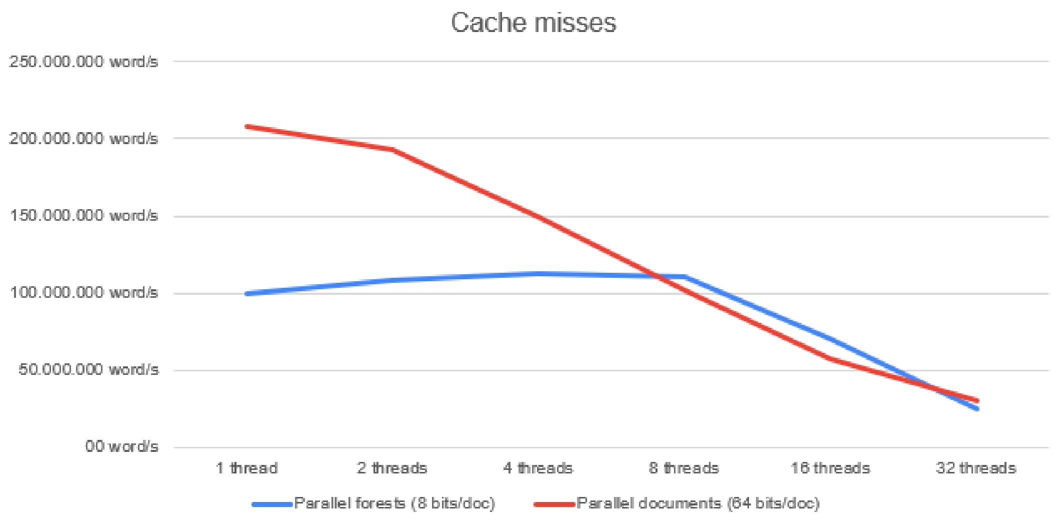
Graph 5: Cache word misses per document of different algorithms

Graph 4 and Graph 5 represent the number of cache misses of the algorithms executed on a single thread. The first one plots the misses per second, while the second one plots the misses for each document.

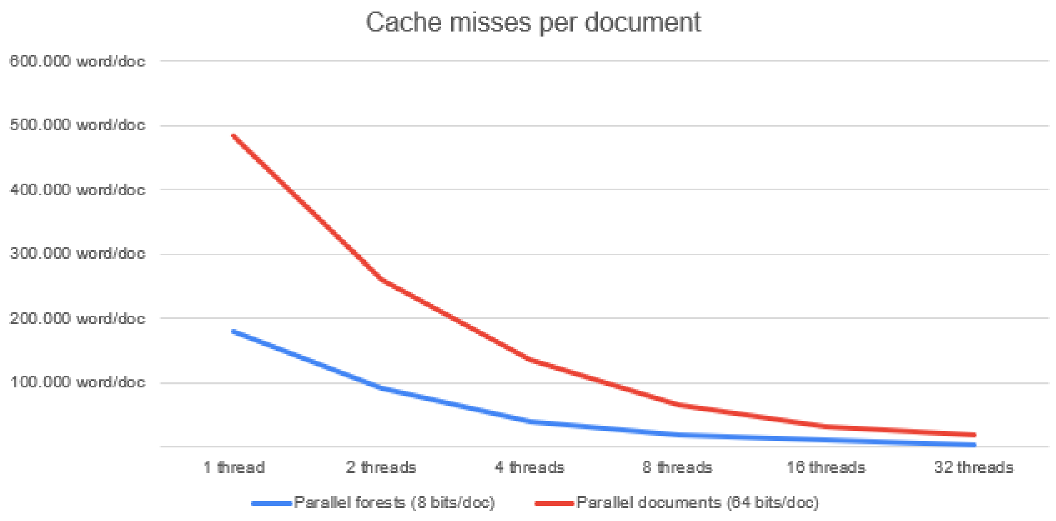
Like we expected the number of misses per document for `MergedRapidScorer` is less than the one for `LinearizedRapidScorer` which is lower than `EqNodesRapidScorer`.

It's interesting to note that for SIMD algorithms the number of misses per document is always less than the non-SIMD ones. This confirms that computing more documents at the same time reduces the overall amount of data to read from memory, since, for example, the thresholds are read once and used for multiple documents.

6.3.2 Multithreading comparison for SIMD512



Graph 6: Cache word misses per second on different number of threads



Graph 7: Cache word misses per document on different number of threads

As we can see in Graph 6 and Graph 7 increasing the number of threads decreases both the absolute and the per document cache misses. This is due to the fact that parts of the data is shared by different threads. As expected, the number of misses parallelizing the forest is lower because it operates on a smaller number of documents at the same time, leading to more cache hits.

6.4 All results

In this chapter we've shown only the most interesting results. The complete list of raw measurements is available in this spreadsheet.