

Class: COMS 4995: Parallel Functional Programming.

Name: Matthew Trent.

Columbia UNI: mrt2177.

Filename-Like Name: bmw (this stands for Block-Max WAND).

Purpose: Communicate with TAs and the professor about the algorithm I'm going to improve.

Groupmates: I got permission to work solo from the professor on Ed.

Haskell Project Proposal: Block-Max WAND (Denoted "BMW")

Overview

Mainstream search engines are forced to answer the same question constantly: given a few keywords, which k documents are the most relevant? Putting aside the production complexity coupled with search, there are three primary baseline implementations I'll investigate for my project that answer this question: exhaustive search [1], WAND (Weak And) [2], and finally Block-Max WAND (BMW) [1] — each building on the last. Thus, my final goal is to have a parallel high-performance Haskell implementation of the final layer: BMW.

Technically, from my research, there exist further enhancements/variants, namely Variable Block-Max WAND [3], Block-Max MaxScore [4], BlockMax WAND L-S (Longer-Skipping) [5], alongside assorted variations of the algorithm that trade accuracy for performance. However, these aren't my primary objectives as I want to focus on the core, standard industry algorithm. Thus, time-dependent, I may foray into assorted variations of BMW, but don't expect to.

Why this Algorithm in Particular

I've been fascinated by the problem of search for a long time. I wrote my thesis on creating a decentralized peer-to-peer search engine in higher dimensional space, experimented with improving recommender systems by creating modified HNSW indexes, and currently lead search, discovery, and recommendations at Sorce, a San Francisco-based startup. Thus, this problem — being in the search space — highly piqued my interest and aligns well with my industry experience.

The Sequential Algorithm

Building Blocks: An Inverted Index

Of the three algorithms, all operate upon an inverted index. This means, for every word w , there is a list of associated documents d that contain it paired with a score. To illustrate this, I'll create an example similar to that from pages 166-167 of a UMass textbook [6] I referenced:

$w_1 = \text{"dog"}$	$d_1, \text{score} = 1$	$d_2, \text{score} = 0$	$d_3, \text{score} = 0$	$d_4, \text{score} = 0$
$w_2 = \text{"fetch"}$	$d_1, \text{score} = 2$	$d_2, \text{score} = 1$	$d_3, \text{score} = 1$	$d_4, \text{score} = 1$

$w_3 = \text{"ball"}$	$d_1, \text{ score} = 5$	$d_2, \text{ score} = 0$	$d_3, \text{ score} = 0$	$d_4, \text{ score} = 2$
-----------------------	--------------------------	--------------------------	--------------------------	--------------------------

Figure 1: Inverted Index. Technically, 0-scored items aren't normally included in each list, but I've included them for illustrative purposes here (pages 166-167 [6]).

Here, we show how an inverted index stores documents. Instead of mapping each document to what words it includes, we instead map each word to the list of documents (called a "posting list" [6]) that include the word, where each document entry includes a score that denotes to what degree the associated word showed up inside each document. In trivial cases, that score value may be the raw appearance count within each document. In more complex systems, such as in web crawlers, this score is often the output of an algorithm that is used to determine how important each word is to the page. For instance, "dog" inside an "h1" HTML tag might be worth more than "dog" inside a "p" tag. Of course, production systems are much more complex — largely due to scale, dealing with bad actors, and "intelligence" (for the latter, I highly recommend looking into how Google's backlink system works [7]).

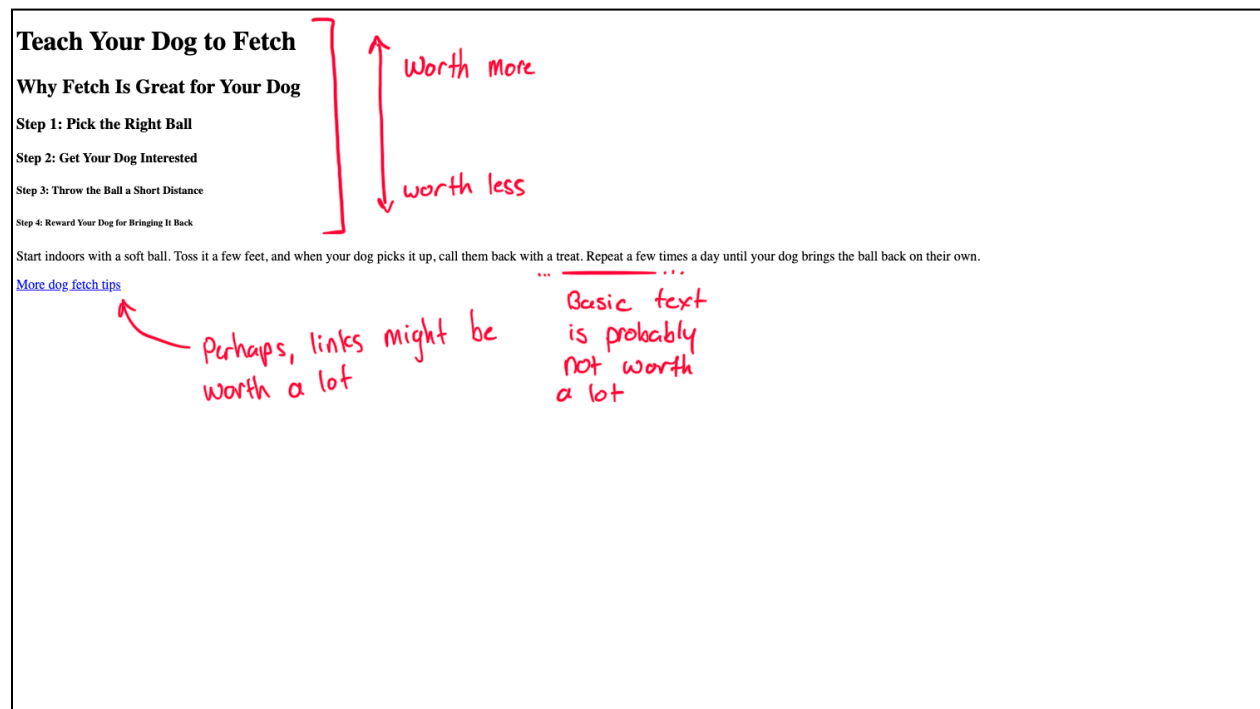


Figure 2: Example HTML page and how "more important" elements may contribute to a higher document score. I wrote my thesis on search that utilized an advanced version of this for document processing — there's a lot of "alpha" in this scoring system.

To actually use the inverted index from Figure 1, in plain language, one might search for "dog fetch" (someone learning to play fetch with their dog), so we pull the posting lists associated with those two keywords (w_1 and w_2) and go document-by-document, summing the score of

each document across all keywords. We then take the k documents with the highest scores. From our running example, if we were to set $k=1$, we'd return d_1 .

Core Algorithm Assumptions

The algorithm I will be optimizing operates under the assumption that this inverted index is already built. For my project, I intend to maintain this assumption and not include indexing time as part of retrieval nor within my runtime calculations. This is because — based on my industry experience alongside assorted research — indexing is often done offline, and hence it being optimized loses in precedence to perfecting the pure BMW hot-path search.

We also assume the inverted index has arranged the documents in a sorted (ascending) order. By this, I mean each entry w_n includes its list of documents arranged in a way akin to $d_1, d_2, d_3, \dots d_n$. This is important for the algorithms later, and is a safe assumption because when we're building the index, we have the choice on how to choose to order, so it's "free" to select this paradigm during construction.

How Baseline Exhaustive Search Works

We'll assume the inverted index from Figure 1 with search "dog fetch". Exhaustive search works as follows:

1. Select the posting list attached to each matching keyword. We'll select "dog" and "fetch" lists.
2. Place a cursor at the start of each list.
3. Find the smallest document ID that a cursor is pointing at. This is the next document to score across each matching word (works because IDs are sorted in ascending order, as described earlier).
4. Score the document by summing each of its matching words' scores. For d_1 , that is $1+2=3$.
5. Propose the document to a min-heap holding the top k documents you want to search for. If the heap has fewer than k entries, this document will always be accepted. Otherwise, it replaces the lowest-scoring document in the heap, but only if its score is strictly higher ($>$, not $\geq$).
6. Repeat steps 3 through 5 until every cursor is out of list.

The main problem with this is that — while correct — it is terribly slow because it runs through the entire length of each posting list per search. This becomes a *huge* problem in large systems, especially when some words are very common and have tens-to-hundreds-of-millions (or more!) of entries in their posting list. An example of some of these huge lists might include posting lists associated with terms like "the", "it", or "a".

How WAND Improves Upon the Baseline Exhaustive Search

WAND (Weak-And) [2] improves upon this base exhaustive algorithm. Pre-search, it stores one additional number per word: the highest score in its posting list. In our running “dog fetch” example, it would store “dog” = 1, “fetch” = 2, and “ball” = 5. Now, the algorithm can exit prematurely if it determines there is no remaining document that can *possibly* beat the k-th best scoring document already inspected.

In our “dog fetch” example, once d_1 has been assessed a score of $1+2=3$, we know that “dog” documents are trivially finished because there is only one, and since we know the “fetch” posting list’s highest possible score is two, and two can’t beat three, we can stop without having to investigate every single “fetch” entry. This massively saves on the amount of documents we must score to get the top k. WAND can also skip forward. If a common word like “the”, “a”, or “it” can’t win on its own, WAND jumps the cursor straight to the next document containing a rarer query word, never scoring the documents in between. This is highly performant because it reduces the number of raw comparisons needed greatly.

How BMW Improves Upon WAND

BMW [1] notices the improvements WAND [2] gives, but asks another question: “what if a word’s single highest score is misleading?” The best way to understand this is by analogy. Say you’re looking for the k most expensive houses in California, walking through each listing in order of listing number. WAND is only aware that “the most expensive house in California is \$17 million”, so, until you’ve found the k houses priced above every remaining possibility, you must inspect every single listing. BMW approaches this differently. It splits the listings into pages of ten consecutive listings (configurable, ten is just an example), and stores the most expensive per page. This means, when you reach a page that says “the most expensive house here is \$5 million”, and the k-th most expensive house you’ve found is already above \$5 million, you can skip the whole page without needlessly inspecting each individual listing. In short, you only stop on pages whose top price could beat your current k-th best. With this said, the actual algorithm has blocks instead of pages, documents instead of houses, and a score instead of a price.

How I’ll Parallelize

To reiterate and be clear: all these will apply to BMW [1], not exhaustive search nor WAND. Furthermore, I will optimize BMW to be as fast as possible sequentially first *then* begin parallel optimizations.

First, I’ll get the trivial case out of the way: I can parallelize inter-query. This is the simplest form of parallelization and will look like running `search(“foo”)`, `search(“bar”)`, and `search(“baz”)` at the same time.

Next, I’ll talk about intra-query parallelization (the “real deal”), namely making each independent search faster. The following are techniques I believe may be applicable. During implementation, I expect it to both grow with new techniques and shrink with techniques I determine don’t work as well as expected.

1. Split BMW's documents into ranges by document ID (so each separate range has a unique set of document IDs), each with a whole number of BMW blocks, so no block is "split" across ranges. Then, run a separate BMW over each range in parallel with `parList` [19], so that each finishes nearly (hopefully) N times faster, where N is the number of ranges (assuming N is $\leq$ the number of cores). Further, I'd use "rdeepseq" [20] to ensure each range fully executes, not doing anything lazily. Notably, it's worth knowing the more ranges I break this problem into, the less each one will skip, thus, I think the upper bound in speedup is less than N times faster in non-contrived cases. Finally, I would then compare each result at the end from each sub-BMW.
2. Tune the number of ranges from step (1) (using "parListChunk" [19] to split the work or "parList" [19] — I will have to empirically check both). The more ranges I have, the less skips I do. But, the more ranges I have, the more general comparison work (CPU burn) can be parallelized. I will need to measure this tradeoff and find the optimal point. I expect the number of ranges to be heavily influenced by the number of CPU cores the runner's machine has. Further, I expect some ranges to take longer than others. Thus, it's possible one core can do two separate sub-BMW ranges in the same time a different core executes a single, long sub-BMW range.
3. From (1) and (2), possibly skipping full ranges of sub-BMW algorithm execution if I can determine an entire range cannot beat out an already-executed BMW range for the set of query words. Specifically, for each query word, if I take the highest block maximum inside a range and those add up to no more than the k -th best score found by another completed range, no document in that range could be included in the top k , so I can skip its execution. In effect, this is applying BMW's own "block check" but with an additional layer of abstraction across parallel executors.
4. Take a greedy approach to range-run-order in terms of how promising each range is. This pairs with (3), because if we can optimize to have a high number of sub-BMW executions go first that find "winning top k results", we can maximize the number of future ranges we can skip execution of entirely. To do this, I would use Par Monad [22]. Specifically, "runPar", "spawn", and "get" to start the most promising ranges first, await their results, then pass the running k -th best scores to the next ranges. This break-up would still be core-dependent, and would need a lot of tuning. For instance, how many ranges should be a part of each "wave" in this method?
5. Though I'm not sure about this optimization (it requires empirical testing), perhaps we could greedily run a few ranges that we presume will both (a) execute quickly and (b) output elements likely to be in the top k set. Then, after this, we can pass these results to each sub range so that they can skip more than if they all started fresh. This is very similar to (4).
6. For the edge case where a user's query is both (a) very big and (b) matches a lot of words in a single document, we could parallelize the addition of scores itself. I assume the overhead of creating a spark will be larger than the benefit of this in almost every case. So, this, again, is worth testing.
7. Instead of waiting for ranges to finish from (4) or (5) *then* starting the next wave of results, we utilize a "live scoreboard"-type technique. All ranges run at once in their own

threads via “forkIO” [23] and “share messages” via the “IORef” [24] single mutable variable, which the documentation denotes is a way to have “mutable references in the IO monad” that you can share and write to via “atomicModifyIORef” (note the ‘ symbol, the documentation includes it, but I haven’t tested with this so I’m assuming it’s correct) [24]. This would effectively mean every range starts at once (core count dependent), and all read/write from the single data source of the top k-th result score threshold. Then, this would in theory allow them to as-aggressively-as-possible skip within their own BMW blocks when a new high scoring item is found by another range. I want to measure this approach against (4) and (5), as they are all quite similar.

Additional Optimization Techniques

I’ve done a lot of software optimization in C and assembly before, and know how to utilize things like loop unrolling, SIMD, register hints, floating-point-to-integer optimizations, and more (I list more techniques enumeratively in the Evaluation section). However, for Haskell specifically, I believe a lot of this does not apply. I’ve been researching GHC (Haskell’s compiler) and it appears a lot of the high-level code optimizations can work, namely things like floating point optimizations, lookup tables, and branch reduction tricks (bit math), however even these don’t seem extremely applicable in BMW optimization. Further, a lot of the lower-level optimizations are simply not possible, or feature support is limited or experimental (i.e., SIMD) [13]. Alas, I’ve included this section just to highlight that I’m going to attempt to use these advanced techniques, despite them not being the primary focus of this class.

Additionally, I’d like to mention trivial enhancements. For instance, I could do the “jump to next document” search using linear search instead of binary search. Or, I could use incorrect data structures for storing the posting lists. Things like this I don’t really consider “improvements” if I improve them. Doing them correctly is the baseline.

Potential Future Directions

There are two main future directions I could take this project:

1. Sacrificing accuracy for speed. Just from being in the search space long enough, I know a lot of algorithms gain huge speedups from being approximate (i.e., HNSW indices for ANN instead of kNN semantic search). I bet this problem could benefit in the same way.
2. Researching further into existing variations of BMW that claim assorted improvements, such as Variable Block-Max WAND [3], Block-Max MaxScore [4], and BlockMax WAND L-S (Longer-Skipping) [5].

Acquiring Input Data

Webpages for Inverted Indexing

Because of my background building websearch, I wanted to apply my algorithm to the same domain. Ergo, I know Common Crawl [8] has huge hosts of pre-crawled websites that we can use as our documents for this algorithm. Additionally, it's free and open, and specifically says it "can be used by anyone".

It holds 2.14 billion webpages across 33 million registered domains (from its August 2026 crawl alone [9]). The main data they collect — and I'd use — from this corpus per webpage is:

1. Raw HTML. This would allow me to create my own scoring algorithm.
2. Page title, description, and other core SEO attributes (extracted from the raw HTML).
3. Page language. This is important because it allows me to pre-filter when creating the index to English webpages only. This is critical for ease-of-demonstration. Further, it's intuitive that all approaches we use for English will generalize well and are mostly language-agnostic (technically true with small asterisks such as character encoding, diacritic folding, etc.)

I would aim to download a million usable pages. Because it's not part of the core algorithm — and because I'd need it to be fast — I'll likely write the index creation in Go or Rust (languages I'm most familiar with). If this is an issue, please raise it with me!

Real User Queries to Test Against Completed Algorithm (to Benchmark)

Beyond webpages, I need a set of data that I can use to run real user searches against my Haskell parallel search algorithm. I could use basic words, a dictionary, or a set of 5-10 manually typed search terms. However, I found that Microsoft offers a myriad of search queries as part of their msmarco project [10]. Further, its stated intention is to help with top-k retrieval systems, and it has 367,013 queries to aid in that. I will ingest these and use them to test my algorithm across a wide spread of searches to ensure I don't "overfit" when testing, which would be likely otherwise.

Evaluation

Post-implementation, I'm sure I'll have a much grander understanding of what I need to evaluate/check. However, my tentative thoughts are as follows.

In addition, I have a 48GB M4 MacBook Pro with 14-cores (cores tested via ``sysctl -n hw.ncpu`` command), so I'll likely benchmark on that. However, it's worth noting that some of the cores are dedicated to being "efficiency cores", so I expect to plateau at around 10-core performance on my machine [12].

Correctness

Correctness is two-pronged: first, each of the following should always, deterministically — if given the same input — return the exact same output:

1. Exhaustive search.
2. WAND.
3. BMW.

Second, the “obvious case” is that each algorithm should output the correct result. I will need to verify this manually myself on smaller cases. For larger cases, I need to ensure my BMW variant returns the same results as the exhaustive (obviously correct) case.

Wall-Clock & Speedup

I want to measure simply how fast I can push my parallelized algorithms. That is wall-clock time. Then I want to be able to check the speedup. That will technically be the more key metric to track for parallel improvements. Namely, it will be the quotient of the wall-clock time on 1-thread over N-thread implementations.

Core Utilization & Spark [17] Statistics

The main part of this evaluation section is not to applaud my algorithm for being a certain degree faster, but to interrogate it for *why* it's not as optimal as it could be. For instance, if I have speed Y at 3 cores, why do I not have speed 2Y at 6 cores? Further, a quick note is that my Macbook (which I'll likely test this on) has the same number of cores as threads, i.e., there is one core per thread [15].

I'll utilize Haskell's runtime statistics [14] to dive into this. Namely, GHC's docs (the Haskell compiler [21]) say “ThreadScope” “interprets the event log to produce a visual parallel execution profile of the program” [14]. This will be extremely helpful for explaining if work that could've been done in parallel, was done in parallel.

To be specific, I believe my goals are to ensure:

1. All cores that can be used need to be used (I say “can be used” because some things cannot be made parallel, and that's OK).
2. Is the work I do in parallel *worth* running in parallel? It needs to be. Coming from a Go background, I know there is an overhead to spinning up a thread in parallel, thus they're not always worth using. Intuitively, this must hold for Haskell (or more specifically for its sparks, which “[are] a pure computation which may be evaluated in parallel” [17]). An example would be summing a large sequence of numbers together. The serial path might perform better than the “fully parallel” approach of booting up a new spark to sum every single pair of numbers in the list. Thus, the “optimal” technique may be to boot up N parallel sparks to where N is the cardinality of the list we sum divided by some constant (to create batches, each running in parallel).
3. Ensuring Haskell's garbage collector isn't pausing threads needlessly. Because it doesn't run fully concurrently, there is a potential loss here [16].

These won't change my speedup numbers, but they'll explain them. That will be extremely important during development.

Amdahl's Law [18]

This reports, given the fraction of your program that can be run in parallel, the best speedup on s cores:

Amdahl's law gives the theoretical **speedup** in **latency** of the execution of the whole task *at fixed workload* W , which yields

$$S_{\text{latency}}(s) = \frac{TW}{T(s)W} = \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}}.$$

Figure 3: Amdahl's Law from Wikipedia [18]. T is the time a program takes to run serially. $T(s)$ is the time it takes in parallel on s cores. W denotes the units of work being done. S_{latency} denotes the theoretical speedup of a program from running on s cores. p is the fraction of your program's run time that can run in parallel.

I will use this to estimate p — the fraction of my program that actually runs in parallel. After measuring my speedup from s cores, I'll be able to solve for it using the formula above.

Other Assorted Performance Techniques

As I've mentioned above, I've done low-level hardware improvements before in C and Assembly on specific processors [11]. This involved techniques (in implementation and/or research) like loop unrolling, inline assembly, SIMD, floating-point to integer conversions, fixed point arithmetic opposed to floating point, compile-time lookup tables, processor-specific NEON vectors, register hints, branch reduction, bitwise and modulo checks, software pipelining, quantitative normalization, processor-specific compiler flags, and much more.

This is all to say I know these techniques exist. However, I'm quite sure they do not exist nearly to the same extent in Haskell. However, it would still be neat to see if I can implement any of these additional more advanced software optimization techniques and run the totality of my benchmark suites before and after integration into my algorithm.

References

- [1] <https://research.engineering.nyu.edu/~suel/papers/bmw.pdf>
- [2] <https://dl.acm.org/doi/epdf/10.1145/956863.956944>
- [3] <https://dl.acm.org/doi/epdf/10.1145/3077136.3080780>
- [4] <https://www.elastic.co/search-labs/blog/more-skipping-with-bm-maxscore>
- [5] <https://par.nsf.gov/servlets/purl/10171670>
- [6] <https://ciir.cs.umass.edu/downloads/SEIRiP.pdf>

- [7] <https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X>
- [8] <https://commoncrawl.org/>
- [9] <https://commoncrawl.org/blog/august-2026-crawl-archive-now-available>
- [10] <https://microsoft.github.io/msmarco/Datasets.html>
- [11] <https://matthewtrent.me/blog/a/ccm>
- [12] <https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/>
- [13] <https://github.com/agrafix/vector-simd>
- [14] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/runtime_control.html
- [15] https://www.cpu-monkey.com/en/cpu-apple_m4_pro_14_cpu_20_gpu
- [16] https://www.reddit.com/r/haskell/comments/hogbqb/why_cant_haskell_have_a_completely_concurrent_gc/
- [17] <https://wiki.haskell.org/Parallel/Glossary>
- [18] https://en.wikipedia.org/wiki/Amdahl%27s_law
- [19] <https://hackage-content.haskell.org/package/parallel-3.3.0.0/docs/Control-Parallel-Strategies.html>
- [20] <https://smunix.github.io/chimera.labs.oreilly.com/books/1230000000929/ch03.html>
- [21] <https://www.haskell.org/ghc/>
- [22] <https://hackage.haskell.org/package/monad-par-0.3.6/docs/Control-Monad-Par.html>
- [23] <https://hackage-content.haskell.org/package/base-4.22.0.0/docs/Control-Concurrent.html>
- [24] <https://hackage-content.haskell.org/package/base-4.22.0.0/docs/Data-IORef.html>