Software Development for Information Systems

Skouris Emmanouil Nikolaos **sdi1700142**
Kalamakis Antonios **sdi1800056**
Καφρίτσας Χρήστος **sdi1800072**

Department of Informatics and Telecommunications, University of Athens

Abstract
Our assignment was to implement a solution for the problem posed in the 2013 Sigmod world programming contest. Namely, an “inverted search-engine” optimized to match a relatively small number of static queries to a large and ever-changing dataset of documents. Our solution is based on BK-Trees and HashTables utilizing a multithreaded Job Scheduler. In this report we will analyze our solution’s performance and compare the speedup of our various optimizations, finally we will propose some further optimizations that we couldn’t perform due to time constraints.

1. Implementation

1.1 Multithreading

For updating our search-engine into a multi-threaded search-engine we didn’t have to change multiple things inside the search-engine’s core implementation but we had to adjust some functions to divide the information so it runs in multiple threads. The Job scheduler’s implementation is able to run different types of jobs. For example, a worker thread could run an addQuery and when finished, it could run a MatchDocument job.

1.2 Job scheduler

Our implementation is based on a job scheduler for the threads that at the start up creates all the threads that the program is designed to use and keeps them waiting until there is a job available to be run. The sleeping thread gets unblocked automatically when there is an available job, so it doesn't have to busy-wait and consume resources by being on an infinite loop. For that reason in some cases the previous operations need to be completed before starting the next one. To achieve that we have implemented a function inside the job scheduler that keeps the parent/main thread waiting until all the threads finish with their current running task.

1.3 Matchdocument-threads

In order to achieve a multithreaded implementation, the match document function, since it was one of the most time consuming functions in this project, we’ve decided that the most optimal way was to break the document’s words into same length vectors of string, that each thread gets and does every calculation and every insertion needed parallely. Then the main thread then waits to finish all the jobs running and combines the output that the threads computed.

1.4. General implementation details

In general, all our searching methods use one unordered map to be able to locate each query with it's queryID. This is helpful because of the fact that a hash table(unordered map) has O(1) computation time for search and insert and since this structure is used by every searching method we need it to be as low as possible. In more detail about the exact searching method, we have created another unordered map that has words as keys and returns an unordered set with all the query ids for the queries that include this word. Here we used an unordered set because we only need the keys of the hash table structure and not the values.

For the edit-distance search method we use, as instructed, a BK-Tree structure that calculates its distance from the Levenshtein distance word distance algorithm.

For the hamming-distance search method we use exactly as the edit distance one, but with the hamming distance algorithm to calculate the distance between words. Here, as requested, we have a vector of BK Trees, with the size of the maximum word length. Each word size has its own BK Tree so it would be more efficient.

In general the BK tree is a parent-child class action. The parent class is a normal BK tree that uses templates. This structure has most of its functions in a recursive mentality. The child class, named BK_entry is a class with overloaded functions for the search and the usage of the BK tree so it will be easier to use. For the search we overloaded the function mainly because the parent-search function is recursive and after testing with the profilers we saw that the program accelerates dramatically if we implement it non-recursively. Finally all of the BK-based search methods have a maximum distance when calculating the distance between the words since if not, there will be an extreme amount of useless loops that delay the execution of the program.

One good thing to mention is that there are many structures that we use a lot in our implementation. The structures vectors, unordered maps, unordered sets, strings and unique_ptr, are designed with a template mentality just so it will be easier to use. These structures are implemented by soft-copying the std structures but with functions and specifications that we need and make the program faster. For example the bud::string structure, since we don't care that much about space complexity, each word that we store as a bud::string class has the maximum word size so we will win some time.

1.5. Optimizations

Writing a program that processes large amounts of data requires us to be mindful of our code's efficiency and speed, that's why we used perf (a Linux profiler) to find bottlenecks and better understand them. Through profiling, we found that one of the most costly operations was a search operation in the BKTree. This operation was being done recursively, calling itself hundreds of times. Our first optimization was to modify the function's parameters so that the values are passed by reference and reduce the number of copies, however with the help of the profiler, and some luck, we determined that the recursive nature of it was the issue and using a loop was significantly faster. Upon making that change we saw an impressive 60% performance improvement. Despite that single change, we still believed there was more time to be saved, the profiler indicated that the next most frequently called and slow function was the edit distance calculation, which was called both during the creation of our data structures and during the query matching. We identified that since in the

competition specifications there was a limit as to how large the matching distance of a query would be (3) we could make the function stop calculating after it reached that limit. This was done only during the BK tree search operations, since modifying the tree requires the true distance between two words. We also found out that the edit distance calculation algorithm was faster if the first given word was larger than the second one, so we are swapping the words in order to always start with the bigger word. Lastly, we saw that a function that separated the words from a large string and removed duplicates was returning an unordered set, which had costly access time compared to a simple vector, which provided another significant time reduction.

2. Development & Testing

For this section, we go through our development process and testing methodology. We present our use of code sanitizers during development, our process or writing unit test and the evaluation environment we run our code on.

2.1 Sanitizers

During development, we employed heavy use of sanitizer tools. Specifically we used an address sanitizer to detect memory corruption, buffer overflows or dangling pointer accesses, a leak sanitizer to detect memory leaks and an undefined sanitizer to detect undefined behavior in the program. Also, we used a thread sanitizer which we found extremely helpful during adding multithreading, since it helps us identify data races and synchronization errors.
Additionally, we used the valgrind error detection tools at some points to cross-reference with the sanitizers' outputs and better identify errors. With the use of the above tools, and the many compiler flags that we used, we can ensure that our program has no leaks, no buffer overflows, no access to dangling pointers, no undefined behavior and no data races. Lastly, we made the interesting observation that combining sanitizers with valgrind results in unexpected and undesired behavior, such as ballooning memory leaks, errors and frozen processes.

2.2 Testing

In order to evaluate and verify our code we wrote various unit tests that we integrated with Github's "Actions" build system. We used the Catch2 library since some of us already had experience working with it and it's API and integration was simple enough to learn for the rest. We focused on testing individual parts of our implementation (mostly data structures and calculation functions) but we also tested the individual functions of our "Implementation" class. Unfortunately the testing library does not have thread support, so we weren't able to test our code in the multithreaded sections. We also had to deal with another limitation in our testing methodology, the interface provided by the competition required the use of global variables, which prevented us from directly testing some functions used to integrate our implementation with the competition's code.

2.3 Evaluation

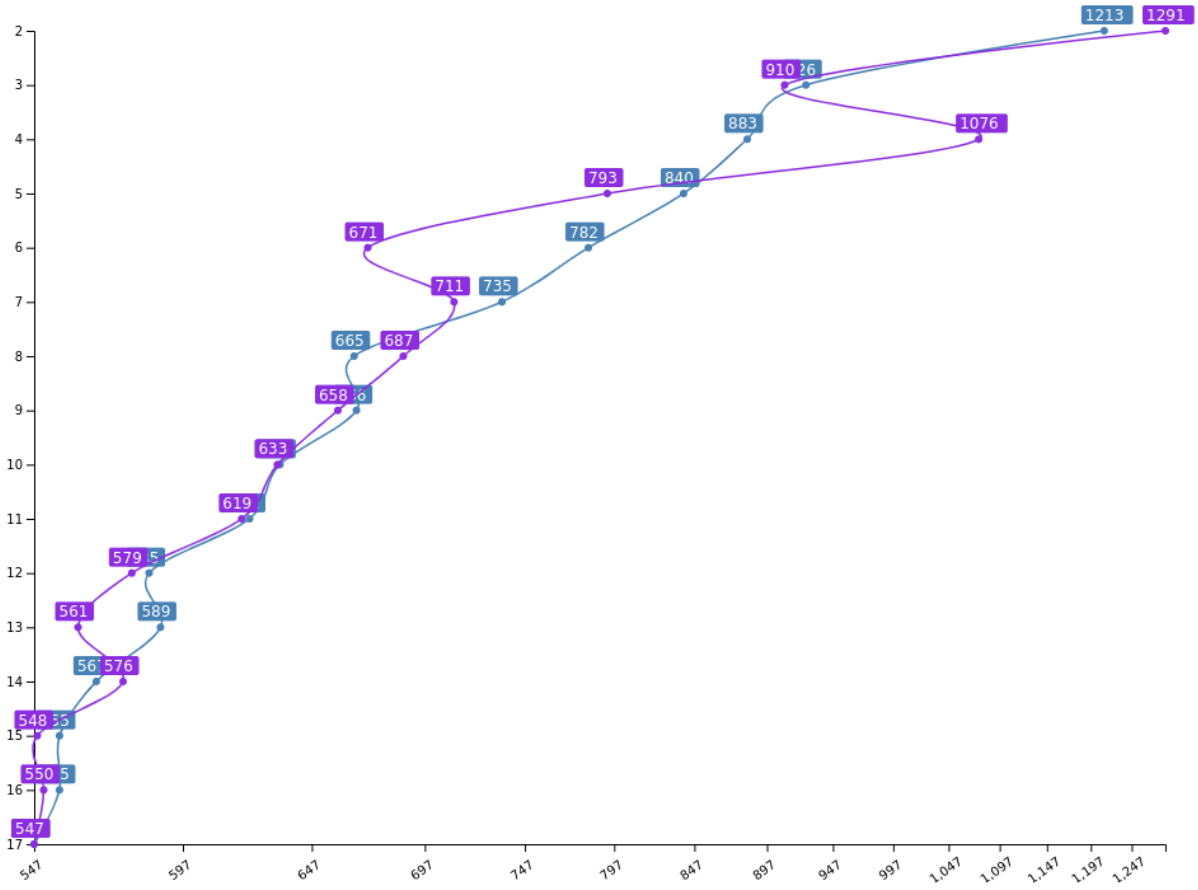
To evaluate our solution we run in multiple times, compiled with optimizations and different compiler toolchains. The test machine we used has 6 cores, 12 threads and 8 GigaBytes of RAM. We ran the program with 2 to 17 threads in order to observe the efficiency of our multithreaded implementation. Each configuration was run 5 times to get the average of the results.

3. Graphs

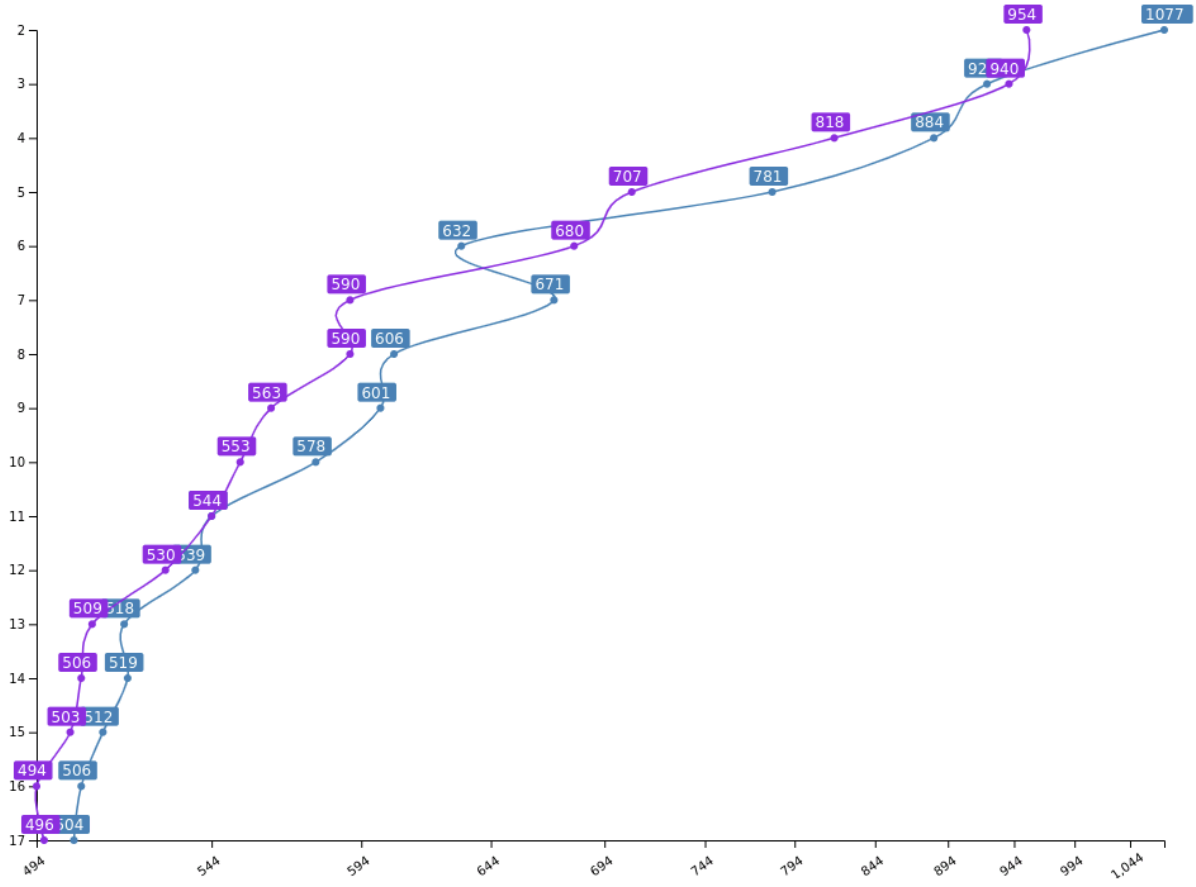
In this section we present some graphs of our program's performance and we draw conclusions from them. The tests were run with the small_test.txt test data. We followed the evaluation methodology we described in section [2.3].

In the following graphs the x axis represents the cores that the program was run on and the y, the time taken, in a logarithmic scale, in order to better present the lower values.

3.1 Comparison between llvm (purple) and gcc (blue)



We can observe that there is no significant difference between the 2 compilers.

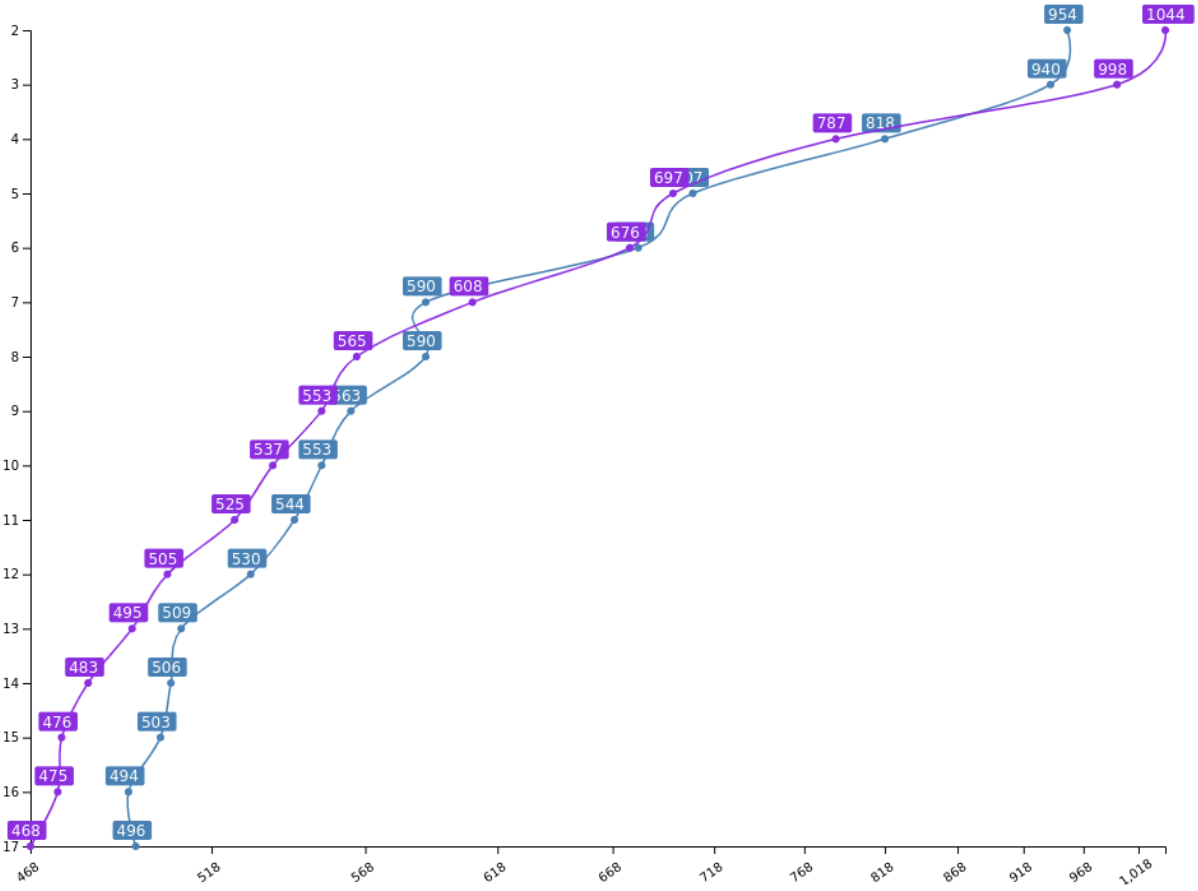


3.2 Comparison between llvm-lto (purple) and gcc-lto (blue)

In this chart we compiled our program with LTO (Link Time Optimization) support.
We can easily observe some improvements over the previous chart.

We can observe that there is no significant difference between the 2 compilers.

3.3 Comparison between llvm-lto (add Query on one thread) (purple) and llvm-lto (blue)



In this chart we enabled LTO and made the add Query function run one single thread.
The improvement in performance is slim, but it is there.
We can observe that there is no significant difference between the 2 compilers.

4. Further Optimizations

In this section we describe some of our theories on how to further improve our solution that we couldn't implement and test due to time constraints. First we present a way to bypass some frequently-used calculations using a caching mechanism, then we talk about reducing the number of thread-blocking functions used (mutexes and condition variables) as well as adding multithreading to less frequently called but still time-consuming functions.

4.1 Cache

During our analysis, we ended up realizing that the most called and time-consuming code segment was the word distance calculation functions. Specifically the Edit Distance function

stood out in regard to our test dataset, taking up a significant 60% of our program's execution time. Evidently, should we manage to speed up some of those calculations, or skip them altogether, we could greatly improve our solution's execution time. Therefore we think that storing pairs of words in a cache structure with fast lookup times would improve the execution time to a great degree. However, there is a point to be made about selecting a way to store and lookup those word pairs in cache that would not require more cpu-cycles than simply calculating the distance in the first place.

4.2 Thread Synchronization

Using multithreading in our implementation greatly improves performance, as evident by [CHART_NUMBER_HERE], however the process of synchronizing the threads is costly since our single threaded implementation runs faster than the multi threaded one on 2 threads. In order to synchronize the worker threads we used a combination of mutexes and condition variables and as a result, we can see that a large amount of time is wasted on thread blocking functions. We believe that some of that time might be better utilized on actual processing tasks if there is a way to reduce the number of thread-blocking functions utilized, or use different thread synchronization methods altogether (for example semaphores).

4.3 Additional Parallelization

For our implementation, we sought after parallelizing the most called functions and the most time-consuming processes (i.e. word matching and structure updates). We think that there are still a number of less-frequently used functions that can be easily parallelized. Adding multithreading to those places might not improve execution time by much comparatively but it is possible to still offer significant reductions while processing large datasets.

5. Running the program

In order to run the program, we will need to compile it and specify the number of worker threads that it is going to use.
So let's suppose that we want the program to use 4 workers and 1 main thread.

The commands are the following:

Create the makefile and other configuration files from the cmake build tool:

```
$ cmake -DCMAKE_BUILD_TYPE=Release -DNUM=4 .
```

After that we will need to build the program. In order to do that we need to execute the following command:

```
$ cmake --build .
```

That is going to create 2 executables, one for the main program and one for the tests.
The former's name is inverted_google and the latter's run_tests.

The run_tests one, doesn't take any arguments, it will only run the tests and produce an output based on the success or failure of them.
So it can be run like this:

```
$ ./run_tests
```

The same can't be said for inverted_google. It needs 2 arguments.
The first one is the name of the test_data file that it is going to read from and execute it's commands.
The second one is the maximum number of seconds that it is allowed to be executed for.
One possible execution of the program is the following:

```
$ ./inverted_google test_data/small_test.txt 60
```

This will create a text file called result.txt. We can open it and see the time taken and the throughput of the program with the command:

```
$ cat result.txt
```

It is also possible (and advised) to run the programs with the valgrind tool.
The above programs could be run like this:

```
$ valgrind ./run_tests
$ valgrind ./inverted_google test_data/small_test.txt 60
```

6. Conclusion

In this paper we presented an implementation of an 'inverted search engine' where the queries are static and the documents dynamic.
The implementation can support searches with exact matching of the word and matching based on edit or hamming distance.
We run our program in multiple compilers, with and without the use of experimental tools for optimization and in multiple threads ranging from 1 to 17.
But, the test data were not particularly big, so the performance gains from enabling multithreading support wasn't that significant.
As a perspective, we plan to compare the performance of our approach with other approaches that use alternative data structuring techniques.