Concurrent Processing of Text Corpus Queries

Radoslav Rábara and Pavel Rychlý

Natural Language Processing Centre
Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
radike@mail.muni.cz,pary@fi.muni.cz

Abstract. The fast evaluation of complex queries on big text corpora is an important feature of corpus managers. The aim of this paper is to apply approaches of concurrent processing to the query evaluation in the corpus management system Manatee. The work contains an evaluation of the query processing speed using various number of cores available, and also a comparison of the length of the source code between the original and the concurrent implementation.

Keywords: query evaluation, concurrency, Manatee, Go, lines of code

1 Introduction

Text corpora are huge collections of texts in electronic form. It is used as a resource of the empirical language data, i.e. words, their meanings and contexts they occur in. Corpora can be employed in many fields of linguistics (morphology, syntax, semantics, stylistics, sociolinguistics etc.) and the corpus managers are primary tools enabling corpus exploration. The corpus managers has to be able to deal with extremely large corpora and to provide a platform for evaluating complex queries, filtering and visualizing results, and computing a wide range of lexical statistics. The speed is an important aspect of the operations because the corpora are usually large – up to billions of words.

In order to speed up the evaluation of the operations, we reimplemented a single-thread evaluation with the concurrent approach within the Manatee corpus management system [5]. The reimplemented system, referred to as the new implementation, was compared to the original system. Not only the performance was evaluated and compared, but also the number of lines of source code was compared.

2 Manatee system

The Manatee system [5] is a corpus manager, it is able to deal with extremely large corpora and the important aspect is the speed of the queries evaluation.

The program has modular design. There is an indexing library for compression, building and retrieving indexes; a query evaluation module with classes for different query operations, a query parser which transforms the queries into abstract syntactic trees, a set of command line tools for corpus building and maintenance, two graphical user interfaces.

The system is based on the text indexing library FinLib that provides procedures for word indexing, corpus storage and retrieving words in form of streams of positions [4]. Manatee has its own query language, which enables retrieving of the word's occurrences.

2.1 FastStream and RangeStream

The implementation of a query evaluation within Manatee is based on streams of tokens or token pairs (representing range of consecutive tokens). The FastStream and RangeStream interfaces (abstract classes) represent token and range streams. Classes which implements them, represent specific operations. The main idea here is to have classes that perform simple operations which can be combined together to perform complex operations.

In the original implementation, these classes are based on the iterator pattern. It means that there is always available only one value from the stream of values. The next value is loaded by calling the `next` method. Once it is called, the previous values are no longer available. Values are always provided in increasing order. After all values are read, an iterator returns a stopper, which is an arbitrary value greater than any value in the stream. Iterator also provides the `find` method to lookup efficiently in the remaining values and the `peek` method to get the following value, which will be returned by calling the `next` method, without proceeding to the next value. More details about the implementation are in [4].

3 The Go programming language

Go, also referred to as golang[1], is a new programming language which has been developed since 2007 as Google project. Go tries to combine performance and security advantages of compiled language like C++ with the development speed of a dynamic language like Python [2]. It was developed by famous programmers such as Ken Thompson[2], author of the Unix operating system, Rob Pike[3], also a member of the Unix developing team, or Robert Griesemer[4], who worked on the design and implementation of the Java HotSpot virtual machine.

The concurrent running functions are called goroutines in Go. It is a coroutine attached to a thread. Multiple coroutines can be attached to a single thread. The attachment is performed dynamically by the Go runtime. The

[1] `https://golang.org/`

[2] `https://en.wikipedia.org/wiki/Ken_Thompson`

[3] `https://en.wikipedia.org/wiki/Rob_Pike`

[4] `https://en.wikipedia.org/wiki/Robert_Griesemer`

attachment of the coroutine can be changed to another operating system thread when a different coroutine attached to the same thread is blocked, e.g. by calling a blocking system call.

Communication between and synchronization of the concurrent running goroutines can be provided by channels. Channels exchange data from the sender to receiver. The communication blocks the sender until the receiver receives the data. In this way, the goroutines provides synchronization of goroutines without explicit locks or condition variables.

Channels can be buffered. The buffered channels blocks the sender only when the buffer is full and it blocks the receiver only when the buffer is empty. More details about Go channels could be found in [3] and many tutorials at the Go site: `golang.org`.

The new implementation was written in Go.

4 Implementation

In the new implementation, FastStream and RangeStream are structures that provide channel as the stream of the positions. The original methods `next`, `peek`, and `find` are no longer provided because the stream of the positions is represented by the channel. Also the operation `find` was removed from the new implementation because the operation was slowing down the application almost in all cases, although it was expected that the operation improves the application performance by reducing the amount of the transmitted data.

FastStream and RangeStream were renamed as FastChan and RangeChan in the new implementation. This naming convention, which uses suffix "Chan" instead of "Stream", emphasizes data transfer over the channels.

In the original implementation, FastStream and RangeStream are interfaces and the classes which implements them, represent specific operations. The new implementation does not have classes implementing FastStream and RangeStream interfaces, but there are functions with FastChan or RangeChan as the return value. The functions create channel, run a goroutine, and return the created channel. The goroutine executes computation and the results are send via the channel, which is closed after the computation is done. This approach is generally known as the generator pattern [1].

4.1 Data exchange between goroutines

An interesting paradox was discovered during testing of the initial concurrent design. The faster computation was expected when the program was running on more than a single core, but the performance was worse. It seemed not only that the computation is not running in parallel, but also that the program is slowed down by runtime task scheduler. Measured results are showed in Figure 1.

The program with the initial concurrent design contained only three goroutines. The program performed the operation of intersection (operation AND)

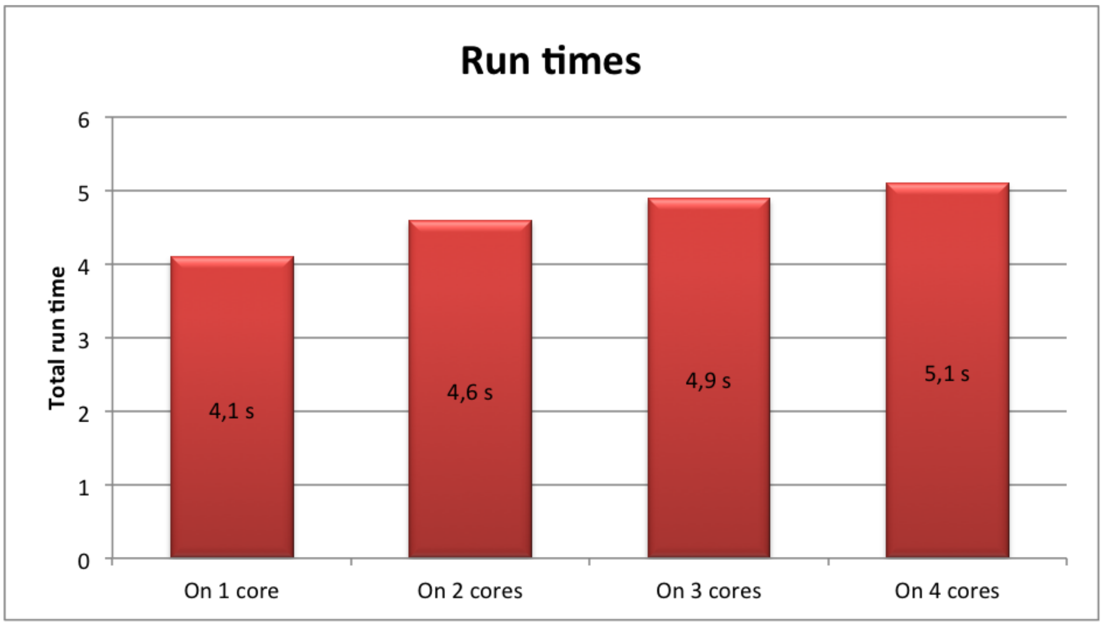

Fig. 1. More cores cause worse performance.

on two channels —- streams of the positions. The streams were loaded from the hard disc in separate goroutines. The relatively slow operation of reading from hard disc could run in parallel with the computation of the operation AND.

The operation AND does not execute time consuming instructions between receiving the data from the input channels. It consists from the simple loop which receives positions from the both inputs and compares the positions. If the positions are equal, it sends the position to the output and loads the next positions from both streams. If the positions are not equal, it loads next position from stream that contains lower position.

Channels use locks to archive synchronous communication and the used design repeately sends and receives data after performing relatively fast computation so the program was slowed down on multiple cores probably because of constantly accessing the channel from multiple cores. Therefore, we tried to send batches containing multiple positions instead of sending single positions.

Sending multiple positions at once causes less usage of channels. Goroutine receives batch of positions, processes the whole batch, and then receives the another batch. The larger the batch is, the bigger delays are between receiving batches from the channel. Disadvantage is waiting for the first batch, which must be filled by a sending goroutine. Advantage can be preloading of batch into the cache memory while adding to or reading positions from the batch.

The results of the program which transmits the batches of the positions instead of the single positions, are presented in Figure 2. The application performs the same operations but the performance is positively influenced by the number of used cores.

In Figure 2, we see that the program speeded up when it run on two cores comparing to the run on one core. It is a proof that data transmission was the issue of the scalability. We can also see that running the program on more than two cores does not cause significant improvement of the performance. It is related to the simplicity of the tested program.

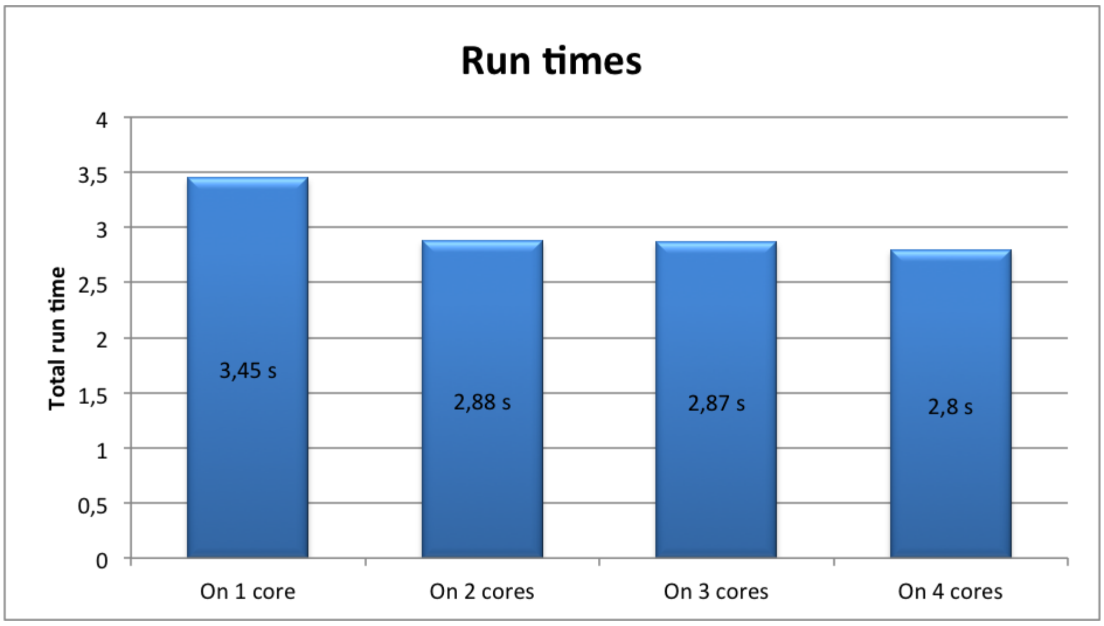

Fig. 2. Sending batch of positions improves performance.

4.2 Comparing the performance

The evaluation of the both implementations was performed on eight processor core server. The original implementation is a single-thread application, so it can use only one processor core. The new implementation has the concurrent design with goroutines, which can run on multiple cores. The new implementation was evaluated with the various number of the cores.

The performance was compared by a simple benchmark that measured time of the evaluation of the prepared queries. The time was measured by the unix `time` command. The prepared evaluation queries are quite difficult and complex because they cover rules of syntactic analysis. The results of these queries are quite big as they cover from 5 to 10 % of the whole corpus. Together the results cover almost the whole corpus. These queries are quite extreme because typical users of the Manatee system usually create just simple queries to find some specified words with some restrictions, which produce much smaller results.

The original implementation evaluated the prepared queries in 4h 29m 24s. The average of the total run (real) times of the queries evaluation are presented in Figure 3.

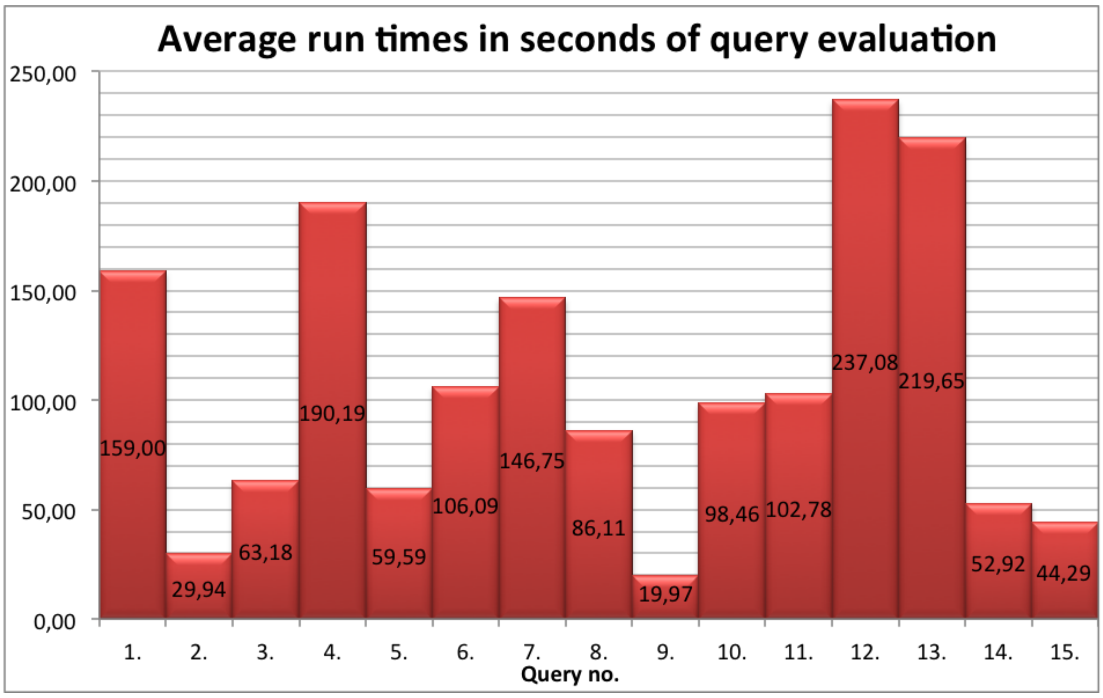

Fig. 3. The average run times of query evaluation by the original implementation.

The new implementation evaluated the evaluation queries in 2h 27m 39s, when it run only with one core. Comparing to the original implementation, the new implementation is faster by approximately 45 %. The avarage of the total run (real) times of queries evaluation are presented in Figure 4.

The Figure shows that thirteen from fifteen evaluation queries were evaluated faster by the new implementation with an average difference of approximatelly 45 %. The fourteenth query had the best improvement by 82 %. The second query had the smallest improvement by 20 %. Two queries were evaluated slower by the new implementation: the ninth query, which was evaluated slower by 116 %, and the eleventh query, which was evaluated slower by 7 %.

The new implementation was evaluated with all possible number of processor cores on the same server, so we can see how the performance is affected when the new implementation is executed on more cores. Measurements are displayed in Figure 5.

The most significant difference is between running on one and two cores, which speeds up the evaluation by 43 %. When the new implementation runs on more than two cores, the performance is better with the each added core, but the cores addition results in slightly less difference of the performance. Only

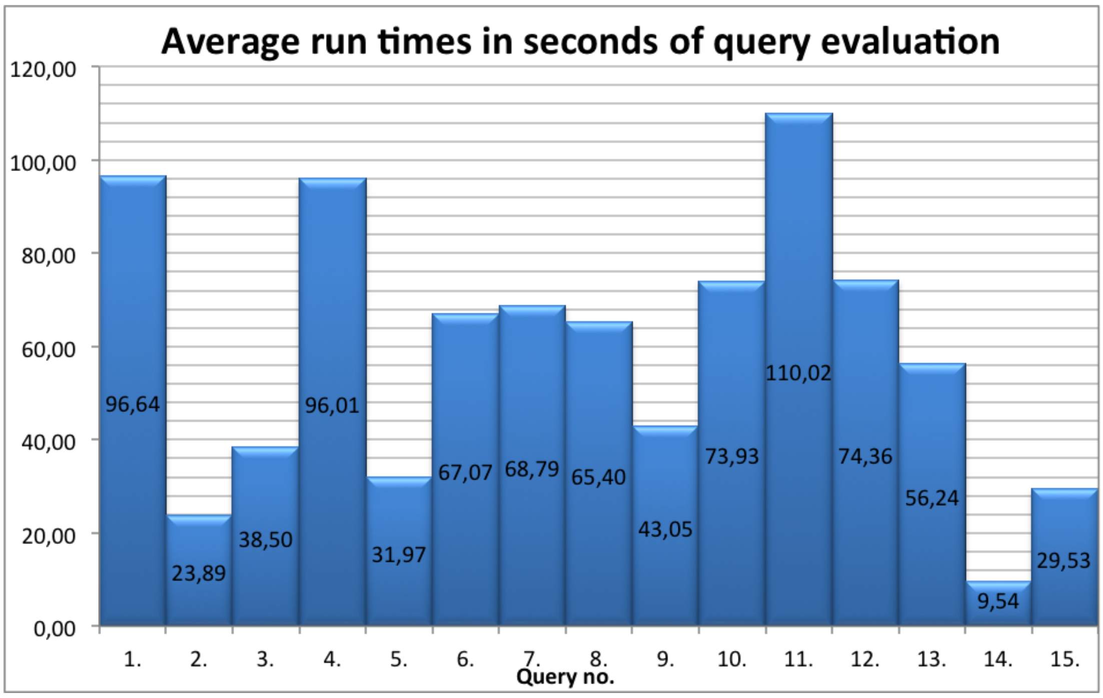

Fig. 4. The average run times of query evaluation by the new implementation.

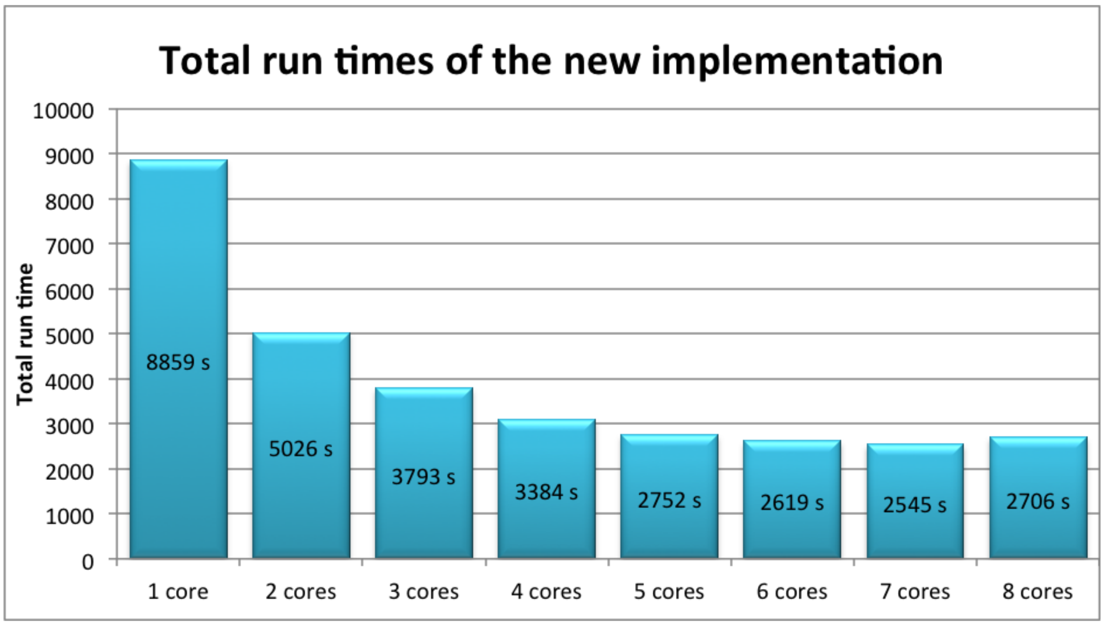

Fig. 5. The total run time of the new implementation running on all possible number of processor cores.

when the new implementation run with the all eight cores the performance was worse than running with only seven or six cores.

The evaluation with three cores speeds up the evaluation by 24 % comparing to the evaluation with two cores and by 57 % comparing to the evaluation with one core. Four cores speeds up the evaluation by 18 % comparing to the evaluation with three cores and by 65 % comparing to the evaluation with one core. Adding the fifth core increases the performance by 11 % comparing to the evaluation with four cores. Runs with six and seven core differ in less than 5 % comparing to the runs without one core.

4.3 Comparing the lines of code

The source code of the original implementation is written in C++, which separates the source code to header files, which contain declaration of an interfaces, and source files, which contain implementation of the declared interfaces. The header files can declare classes, functions or variables. It can also contain implementations of short functions. Working with the source code requires reading from the both — header and source files. Therefore, both file types are included in the source code analysis of the original implementation.

The source code of the new implementation is written in Go, which does not have more types of source code files and it does not require linking between files in the same module. However, it has strict syntactic rules that can prolong the source code.

The source code of the new implementation is expanded by the implemented mechanism of the batch transmission, which provides scalable data transmission used to evaluate queries. The C++ has advantage of the initialization lists[5] which shorts attribute initialization to one line.

The strict syntactic rules in Go forbids one line functions and writing if statement without surrounding block. Go also forbids some common expressions in C++ such as `variable1 = variable2++;`.

The one line functions has its header (return type, function name and parameters) and body (implementation) on the same line. Functions in Go consume at least three lines: the first one consists of a function header and the start of the function's body, the second one contains statement, and the third one is the end of the function's body.

The new implementation does not have all functionality of the original system and therefore, the comparision cannot include the whole source code. Instead, only some units were compared. The units can be divided into two groups:

1. FastStream (FastChan), RangeStream (RangeChan), and structures which provides batch transmisison. The difference between lines of codes indicates how the length of source code was affected by changing from a single-thread to concurrency paradigm.
2. SortedRuns, Read_bits, and Write_bits, which have the same implementation so the difference between lines of codes indicates how the length of

[5] `http://www.cprogramming.com/tutorial/initialization-lists-c++.html`

source code was affected by changing the programming language from C++ to Go.

The results include only the not blank and not comment lines, and only those units of the code were included, which does not provide functionality unimplemented in the new system. The results are presented in Figure 6.

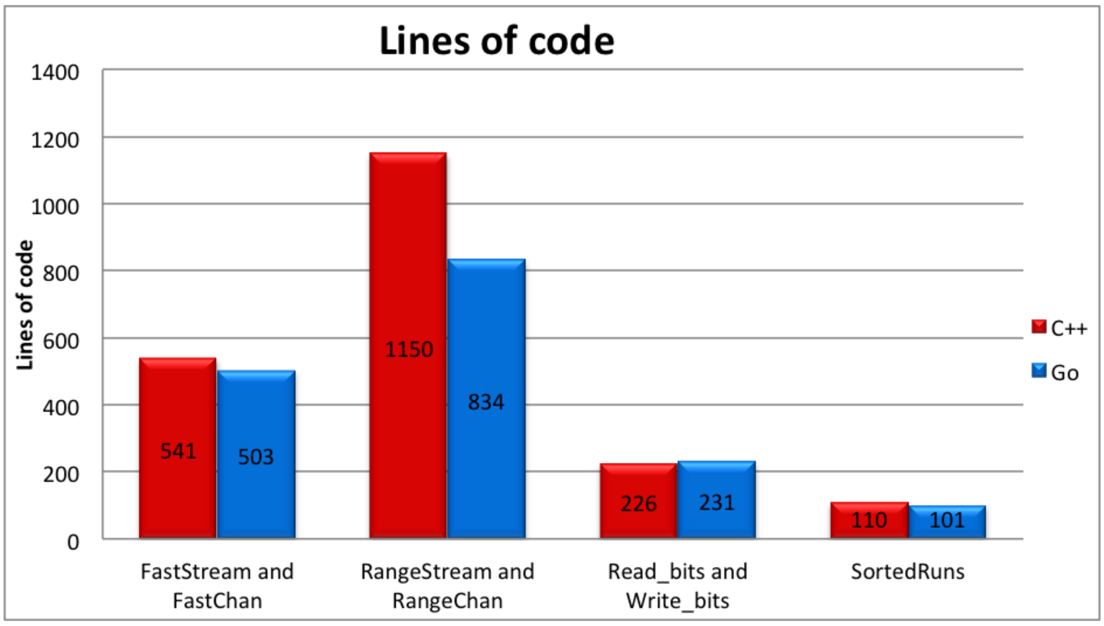

Fig. 6. Graph comparing lines of code of the original and the new implementation.

The number of lines of the reimplemented FastStream and RangeStream interfaces and classes is reduced by 21 %, so the change of the sequential to concurrent paradigm leaded to the shorter source code. The difference could be even bigger if there would not be needed mechanisms of sending and receiving positions as batches, which takes 252 lines of code (19 %).

Reimplementing of Read_bits and Write_bits slightly expanded the source code and even that the reimplemented SortedRuns has less code by 1 %, the reimplemented code of the second group has rather longer code.

5 Conclusion

The performance and the length of the source code were compared between the single-thread and concurrent implementation of the corpus manager Manatee. Manatee is able to deal with extremely large corpora and to provide a platform for evaluating complex queries, filtering and visualizing results, and computing a wide range of lexical statistics [5].

The original implementation (in C++) evaluated the prepared benchmark queries in aproximatelly 4.5 hours. The new implementation (in Go) evaluated the prepared benchmark queries on one core in approximatelly 2.5 hours. The concurrent system has performance better by 45 % on one core. The new implementation was also evaluated with all possible combinations of processor cores. The performance is enhanced by 15.7 % in average by each core of the server and the most significant enhancement by 43 % was between running on the one and two cores.

The new implementation does not have all functionality of the original system so the comparision of the length of code was processed only with some units of the code. The source code analysis points out that disadvantage of C++ are header files and disadvantages of Go are strict syntactic rules and the implementation of the data structures that deal with batch data transmission. The original implementation has 2027 lines of code of the selected units and the new implementation has 1667 lines of code of the appropriate units.

Evaluations proof that the new concurrent implementation has better performance and shorter source code. Therefore, it will replace the original system after the missing functionality will be implemented.

Acknowledgements This work has been partly supported by the Czech Science Foundation under the project GA15-13277S.

References

1. Dolan, R.: Go Language Patterns: Generators. (2009) Available from: `https://sites.google.com/site/gopatterns/concurrency/generators`.
2. Kincaid, J.: Google's Go: A New Programming Language That's Python Meets C++. (2009) Available from: `http://techcrunch.com/2009/11/10/google-go-language/`.
3. Pike, R.: Concurrency is not parallelism. Heroku's Waza conference (2012). `http://talks.golang.org/2012/waza.slide`.
4. Rychlý, P.: Corpus Managers and their effective implementation. PhD Thesis, Faculty of Informatics, Masaryk University (2000)
5. Rychlý, P.: Manatee/Bonito - A Modular Corpus Manager In 1st Workshop on Recent Advances in Slavonic Natural Language Processing. Masaryk University, Brno (2007) 65–70