go-prrrrr: Scalable Parallel Stream Processing Library

Tianjiao Huang

Abstract

go-prrrrr(**Go**lang **P**a**r**allel St**r**eam P**r**ocessing Lib**r**a**r**y) is a scalable parallel stream library in Golang for efficient data pipelines. Golang is known for building efficient large scale server applications with extensive use of asynchronous I/O because of go-routines(green threads) and go-channels(multiple-producer-multiple-consumer queues) are first-class citizens in Golang. *go-prrrrr* makes use of these unique language features to experiment with various runtime optimizations of stream processing.

Contents

Introduction

Stream processing is a popular way to write data pipelines in modern programming languages. It allows programmers to express complex data pipelines in concise and readable manner while leaving rooms for optimizations for the compiler and the runtime. Stream libraries typically allow users to create a stream of data from any data source. A stream can flow into one or more downstream operators. Programmers can chain an arbitrary number of streams and operators together to form complex data pipelines.

Some of the most widely used stream processing libraries are Java's stream, C#'s LINQ, Scala's view, Rust's Iterator, and Python's functional. Most of these libraries will lazily execute the intermediate stages to allow the runtime to optimize the DAG to allow faster execution. These libraries relies on them compiler to perform ahead-of-time(AOT) optimizations or language runtime to perform just-in-time(JIT) optimizations. The primary approach to optimize the stream operations has been JIT since it can make smarter optimizations than AOT thanks to its access to runtime heuristics [6]. However, recent works argue that AOT can achieve better performance than JIT due to JIT's unpredictable behavior and its necessity to make fast action during runtime to achieve better performance [5].

Having access to the syntax tree is important for optimizing streams. However, it is possible to optimize the streams without the compiler support. If the operators do not produce side effects, one can view these stream processing libraries as optimized and user-friendly functional programming libraries. During the era of functional programming, researchers did extensive research on stream processing optimizations [1] [4] [3] [2]. *go-prrrrr* experiments with a few stream processing optimizations in Golang and measures the performance improvement of them.

Architecture

Stream

A stream is a collection of data of same type. Each operator can process one or more input streams and return one or more output streams. Combining various operators and streams will result in a acyclic-directed-graph(DAG), such as fig. 1. Operators that do not generate immediate results, such as `Map()`, `Filter()`, and `Shuffle()`, are known as the intermediate operators. The other operators are known as the terminal operators, such as `Count()`, `ToArray()`, and `ToFile()`. Intermediate operators are lazily executed, meaning that they will not be executed until it is invoked by a terminal operator. Lazy evaluation allows the runtime to perform optimizations at the terminal operators. The ordering of the data in the stream is not guaranteed to allow further optimization. Since each operator could run on a different thread, it's important to reduce the number of message passing between the operators since cross thread communication could be very costly.

Operator Merging

If two neighboring operators has the same number of input streams and output streams and the same type of streams, it's possible to create a new operator that combines the functionalities of the original two operators to reduce the number element passings. For example, in fig. 2, operator `A` and `B` are merged into operator `AB` and the number of message passing is reduced from 3 to 2.

Stream Batching

Instead of passing each individual element through the stream, each operator can pass a batch of elements at a time to its downstream. As shown in fig. 3, stream batching can reduce the number of message passing to how many times the batch size.

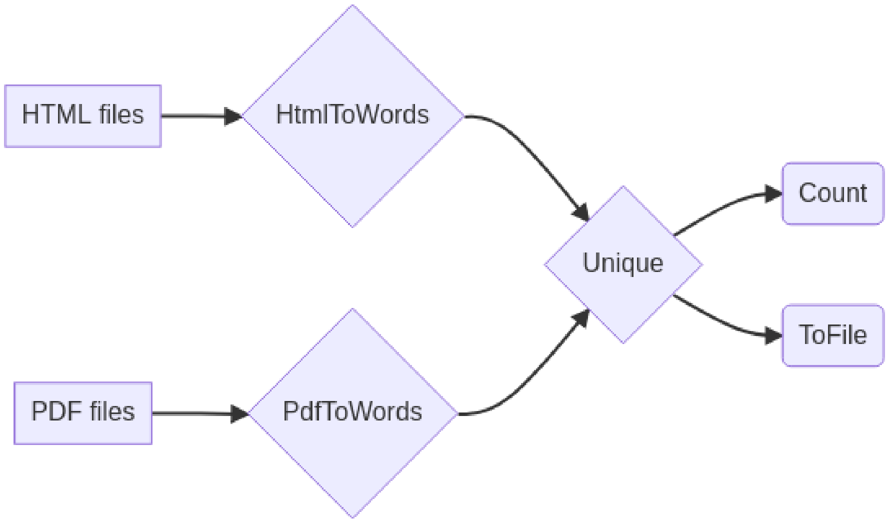

Figure 1: Stream DAG diagram

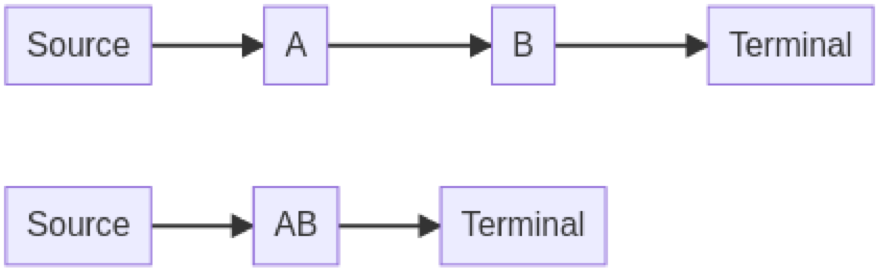

Figure 2: Operator merging

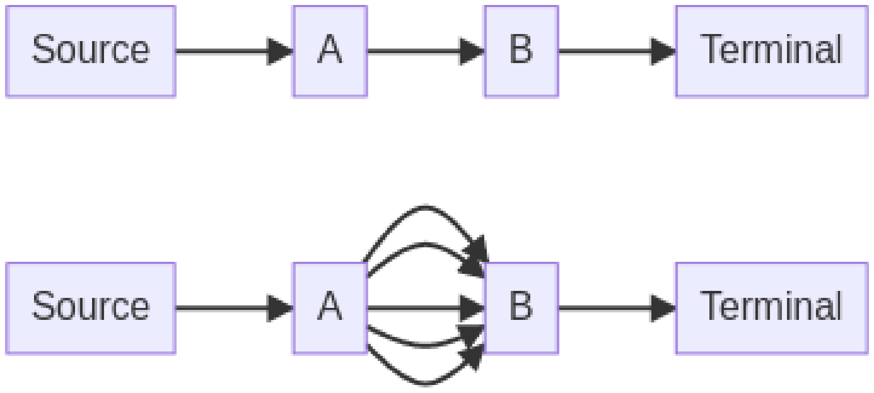

Figure 3: Stream batching

Implementation

Stream Abstraction

For simplicity, we only consider the scenario where operators can only take one input stream and produce one output stream. We also assume that the types of all operators in the same stream are the same. Besides the intermediate and terminal stream operation methods, streams also expose methods to query the worker function and the parent to enable possible optimizations.

Parallel Processing and Data Passing

The worker functions at each operator run in their own go-routines. Unlike most other stream libraries, instead of using an iterator pattern to pass data around, *go-prrrrr* make use of go-channels to allow efficient data flow in the streams. This could be more efficient than Java stream's implementation because no virtual call is necessary to pass data through go-channels. In contrary, it requires $O(N * K)$ number of virtual calls in Java to pass elements around for a stream with N elements and depth of K [5]. The potential downside of using go-channels is that a go-channel is basically a multiple-consumer-multiple-producer queue(mpsc queue) backed by a mutex. If the degree of parallelism is high and the amount of work on each operator is small, the contention of the mutex could be high. This issue could be resolved by using a more efficient mpsc queue or using the stream batching and operator batching optimizations to reduce the number of data passing required.

Stream batching

We batch multiple elements in one slice(array) and pass the slice down the channel to reduce the number of passings. The slices are reused at each operator to reduce the number of heap allocations. Since the number of input elements could be much bigger than the number of output elements, we pack each slice full to maximize the effectiveness of batching.

Operator merging

Since, in our implementation, all operators have the same numbers of input and output and the same data types, all neighboring operators can be merged into one operator. For each neighboring operator, i.e., an operator and its parent, combine their worker function into a new one, and generate a new operator with the merged worker function. The merging operation is done recursively until there is no more operators to merge.

Evaluation

We ran our experiment on a dual-socket Dell R7525 server. Each socket runs a 32-core AMD 7542(AMD EPYC Rome) running at 2.9 GHz. The server has 512 GB of 3200 MHz DDR4 ECC memory. All benchmarks are done in 64 threads with `Map(int.increment)` for the workload of each operator unless specified. All numbers are reported in milliseconds.

Stream Batching

Table 1: Stream batching benchmarks.

Optimization/Batch Size	1	32	64	512	1024
Un-optimized	1847	N/A	N/A	N/A	N/A
Stream Batching	N/A	104	72	45	42

The benefit of stream batching is significant. As shown in tbl. 1, batching with `batch_size=1024` provides as much as 40 times speed up. The speed improvement is expected since most of the anticipated slow down comes from the overhead of message passing.

Operator Merging

Table 2: Operator merging benchmarks.

Optimization/Depth	1	2	4	8	16
Un-optimized	3503	5327	9168	16164	30321
Operator Merging	3518	3596	3671	3813	4424

We are also seeing drastic improvement of applying operator merging when the number of stages is large. For example, in tbl. 2, we achieve an almost 2x speedup for a two-stage pipeline and a 6x speedup. Again, this confirms our theory that the message passing is the main overhead in our implementation.

Merging with Batching

Table 3: Small workload benchmarks.

Optimization/Workload	Light	Heavy
Un-optimized	16265	5327
Batching+Merging	23930	10208

For this benchmark, We ran one big and one small workload with both stream batching and operator merging optimizations enabled. And we set the number of stages of the stream to be 8 and the batch size to be 1024. tbl. 3 uses integer increment, a small workload, like previous benchmarks. tbl. 3 uses a Taylor Series for sine estimation with depth equals to 4 to estimate a relative heavy workload. Our optimizations see performance improvement in both workload with the small workload seeing much drastic improvement. This further confirms that message passing is the main bottle neck.

Conclusions

While it is optimal to have the compilers, whether static, AOT, or JIT, to optimize the stream operations, it's possible for library implementers to improve the performance with stream optimizations. *go-prrrrr* is a proof-of-concept of implementing DAG optimizations on the library side. It implements stream batching and operator merging optimizations and evaluate the performance of the optimized version versus the performance of the un-optimized version. It confirms our theory about the main cost of parallel stream processing is the inter-process communication(IPC) cost.

The source code of this paper and *go-prrrrr* is obtainable on https://github.com /tjhu/go-prrrrr.

Acknowledgements

I thank my friends, especially Zhaofeng and Hongyu, for providing me moral and technical support during the grind.

Bibliography

[1] Coutts, D. et al. 2007. Stream fusion: From lists to streams to nothing at all. *SIGPLAN Not.* 42, 9 (Oct. 2007), 315–326. DOI:https://doi.org/10.1145/1291220.1291199.

[2] Hirzel, M. et al. 2014. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4 (Mar. 2014). DOI:https://doi.org/10.1145/2528412.

[3] Jones, S.P. et al. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. *IARCS annual conference on foundations of software technology and theoretical computer science* (Dagstuhl, Germany, 2008), 383–414.

[4] Kiselyov, O. et al. 2017. Stream fusion, to completeness. *Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages* (New York, NY, USA, 2017), 285–299.

[5] Møller, A. and Veileborg, O.H. 2020. Eliminating abstraction overhead of java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020). DOI:https://doi.org/10.1145/3428236.

[6] Prokopec, A. et al. 2017. Making collection operations optimal with aggressive JIT compilation. *Proceedings of the 8th ACM SIGPLAN international symposium on scala* (New York, NY, USA, 2017), 29–40.