Department of Computer Science, ETH Zurich *Laboratory for Software Technology*

Microsecond-level Scheduling for Apache Spark

External Master Thesis Project for Lukas Drescher

March-September, 2018

Introduction

Apache Spark [1] is one of the most popular distributed data processing frameworks. It offers a unified high-level abstraction (Resilient Distributed Datasets [2]) in a single framework for multiple data processing paradigms such as MapReduce, graph computation, stream analysis, machine learning, and SQL processing. Due to its high performance, convenient/intuitive programming APIs (provide an illusion of a single machine), and support for multiple languages (Java, Scala, Python, R, etc.), Spark is a popular choice for data scientists and developers.

Internally, Spark parallelizes a program or a job by breaking it into multiple independent *tasks* that run in stages. These tasks are scheduled on multiple executor machines to process a part of the data, and finally their outputs are combined to generate the result. As the demand for low latency, interactive data processing increases, over the years, many efforts have been put to make the runtime of a task smaller. Apart from delivering interactivity and high performance, small tasks are also beneficial to maintain the fairness between long running and short running jobs, mitigate stragglers and skew issues, provide fast recoverability in a case of failure, and improve resource utilization [3].

However, making tasks small and scheduling them present some interesting challenges. In order to support small tasks (at milli- and microseconds level), a scheduler not only needs to make hundreds of thousands of decisions per second, but it also has to manage and execute thousands of tasks at a similar rate. A high-performance scheduler design is an active research field where recent efforts [4, 5, 6, 7, 8] have focussed on (i) improving the throughput of the decision process with centralized, decentralized, and hybrid scheduler designs; (ii) increasing interactivity by low-latency scheduling; and (iii) improving the cluster utilization by high-quality task placement decisions. However, not enough focus has not been put to the *task management* and the *control plane logic* that involve launching, cordinating, and executing tasks and then collecting the result [9]. Specifically, in a framework like Spark, there are multiple inefficiencies that stem from (i) the actual scheduler implementation; (ii) runtime/language related overheads [10]; (iii) communication and synchronization overheads; and (iv) data access related overheads. These inefficiencies effectively limits its scheduling throughput to a few thousand tasks per second, while the "core" scheduler might be making hundreds of thousands task placement decisions per second.

Recent advancements in the networking and storage hardware (100+ Gbps RDMA, NVMe flash, PCM memory), efficient I/O in management runtimes (IBM DiSNI and DaRPC stacks), and the availability of high-performance distributed storage solutions (e.g., Apache Crail) now enable us to build a high-performance control path for the scheduler, which is the prime focus of this thesis. Ousterhout et al. already analyzed the possibility of such a proposal five years ago (section 3.3) [3]:

> *"Today's datacenter networks easily allow a RPC to complete within 1ms. In fact, recent work showed that 10µs RPCs are possible in the short term [26]; thus, with careful engineering, we believe task launch overheads of 50µs are attainable. 50µs task launch overheads would enable even smaller tasks that could read data from in-memory or from flash storage in order to complete in milliseconds."*

To the best of our knowledge, we are not aware of any general-purpose system that is scheduling and operating in the microsecond range while scheduling hundreds of thousands of tasks per second. Achieving this performance requires support from the whole stack (devices, runtimes, the framework, operating system, etc.) which is only realized in the last couple of years. As we move towards the *Microsecond-Scale Data Processing Era*, we have evidence that such system design and target performance are now viable [11, 12].

Goal and Scope

The goal of this thesis is to design and implement a Microsecond-level task scheduling framework for Apache Spark. The thesis work should focus on the *mechanism* (how to launch a networked task) rather than the *policy* (when and where to schedule). The thesis output should demonstrate what performance can be achieved without fundamentally altering the design of the Spark scheduler and execution framework (for example, moving from a centralized to distributed scheduling). The target scope of deployment of the scheduler is in a Rack-scale computing, where 10-100s of closely connected machines execute multiple workloads in a high-performance environment.

Tasks

More specifically, the student *should* focus on following tasks to improve the performance of the scheduling framework in Spark:

- **Communication Overheads:** In order to support $< 10\mu$s task launch times, the communication layer in Spark (between the driver and executors) should be optimized for high-performance networks. Here, the focus of the work should be on porting the Spark's Netty/NIO based RPC communication framework to a highly-optimized RDMA based DaRPC implementation [13, 14].
- **Java/JVM Overheads:** Here, the focus should be on removing overheads that emerge from implementing the framework in a managed runtime like JVM. These overheads might come from (de)serialization, data copies, object management, efficient Java/Scala codes, JITing issues, and garbage collection, etc. An example analysis can be referenced here [15].
- **Scheduler Implementation:** Lastly, the effort here should be focused on the implementation-related overheads in the current Spark scheduler code. There may be inefficiencies related to the concurrent data structures used, locking, notification, resource management, unnecessary code in the critical path, etc. General overheads from the inefficient distributed control path for long-running repetitive tasks are discussed here [9].

As we move further in the thesis work, we expect these directions to be revised and new directions to emerge with the input from implementation and performance profiling tools.

General instructions

- The project starts on March 13th, 2018.
- The project should be completed by September 11th, 2018.
- Prepare a thesis draft containing Introduction, Problem Statement and Related Work, and a work plan after a month.
- The progress of the thesis work should be discussed regularly with the supervisors at ETH and IBM Research.

- The deliverables of this work are (1) the source code, (2) a report, and (3) a one page summary.
- The report should be phrased as a scientific essay and in PDF format. It must be in English.
- There should be an oral presentation at the end of the thesis work.

Grading scheme

Grading will be based on the following criteria:

Criterion	Weight (± 0.5)
Implementation (functionality, extensibility, documentation)	3
Report (content, illustration, writing)	2
Presentation (content, illustration, quality of talk)	1
Approach (organization, approaching problems, independence, involvement)	1
Contribution to the state of the art research	1
Completion of all tasks	2

Each criterion will be graded in accordance with the "Merkblatt zur Master-Arbeit in Informatik nach Studienreglement 09".

ETH Professor: Prof. Dr. Thomas R. Gross (trg@inf.ethz.ch)
IBM Research Supervisor: Animesh Trivedi (atr@zurich.ibm.com)

More Papers

Papers about high-performance data processing in systems (single machine or distributed system) that might be of relevence to what we are trying to build.

- SOSP 2017: MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface
- SOSP 2017: Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks
- SOSP 2017: LITE Kernel RDMA Support for Datacenter Applications
- SOSP 2017: ffwd: delegation is (much) faster than you think
- SOSP 2017: Drizzle: Fast and Adaptable Stream Processing at Scale
- SOSP 2017: Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data
- SOSP 2017: Low-Latency Analytics on Colossal Data Streams with SummaryStore

Reading material

[1] Apache Spark - Lightning-fast cluster computing. `https://spark.apache.org/`, 2018. [Online; accessed Mar-2018].

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[3] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 14–14, Berkeley, CA, USA, 2013. USENIX Association.

[4] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[5] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[6] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[7] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 99–115, Berkeley, CA, USA, 2016. USENIX Association.

[8] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 251–263, Berkeley, CA, USA, 2017. USENIX Association.

[9] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 513–526, Berkeley, CA, USA, 2017. USENIX Association.

[10] Martin Maas, Krste Asanović, and John Kubiatowicz. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 138–143, New York, NY, USA, 2017. ACM.

[11] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[12] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[13] DaRPC: Data Center Remote Procedure Call. `https://github.com/zrlio/darpc`, 2018. [Online; accessed Mar-2018].

[14] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 15:1–15:13, New York, NY, USA, 2014. ACM.

[15] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 383–400, Berkeley, CA, USA, 2016. USENIX Association.