Case study: Build native Apache Spark in C++

Abstract

In this report, we present a native implementation of Apache Spark [1] in C++. The original implementation of Apache Spark is built in Scala, which is a managed language based on JVM. Managed languages usually execute user programs by interpreting pre-compiled bytecode and automatically allocate and deallocate memory by runtime garbage collector. In contrast, native languages like C++, Rust, require users to manually handle memory acquisition and release, and the computing units directly execute the compiled assembly generated by ahead-of-time compilers. Therefore, native languages usually provide higher performance than virtual machine based languages. We build Sparkpp [1], our Apache Spark implementation in C++ based on first Spark release (0.5) and test its performance against latest Spark standalone mode. Our evaluation shows Sparkpp can outperform Apache Spark by approximately 2x in tested jobs.

Keywords Apache Spark, Distributed Computing, Native Programming Language

1 Introduction

Managed languages, including Java, Scala, Python, C#, have a virtual machine (managed runtime) for executing pre-compiled bytecode and managing runtime information, including memory handles and object headers. Though they ease the burden of software developers on how to manage the object lifetimes and provide advanced runtime features (E.g. runtime reflection, code stubbing) and Just-In-Time compiling, these languages introduce some cost compared to native languages.

Native languages, like C++, Rust, force developers to handle memory allocation and deallocation explicitly. These languages equip with lite runtime providing OS-specific functions and exceed managed languages in performance.

Apache Spark [1] is a data-intensive distributed computing framework written in Scala. Based on our understanding of native languages, we implement Sparkpp, our proof-of-concept native version of Apache Spark in C++, and get performance improvement compared to highly optimized Spark-2.4.4.

2 Related Works

Apache Spark [1] is a scalable, data parallel computation framework written in Scala, a JVM-based programming language. It leverages the resilient distributed datasets (RDDs) abstraction and lineage tracking to support iterative jobs with high performance.

[1]repo: https://github.com/Airtnp/Sparkpp

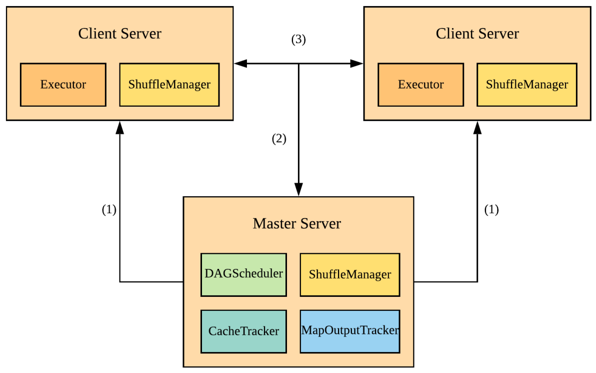

Figure 1. Architecture of Sparkpp. (1) includes the task dispatching, cache status/map output status/shuffle files replies. (2) includes the task result, cache status/map output status/shuffle files requests. (3) The clients exchange shuffle pieces during ShuffleMapTasks.

Native Spark [2] is a native implementation of Apache Spark in Rust and our project inspiration. It argues it provides arguably higher performance than Apache Spark [3]. After careful inspection of the code, we find that Native Spark caches all shuffle outputs in memory, which is different from standard Apache Spark and Sparkpp and highly impacts the performance. Because of this and project deadline limit, we don't evaluate and compare with it in the evaluation section 5.

Thrill project [4] is a C++ framework for distributed data parallel computations. Thrill dispatches same binaries across cluster nodes. Different from Apache Spark, Thrill nodes cooperatively fetch data and do local computations and then push data to other nodes in a MPI [5] fashion. Since our study mainly focus on the case study on native implementaton of Spark, we do not include Thrill in our evaluation part.

3 Architecture

We implement our high-fidelity mimic of Apache Spark: Sparkpp, based on its first release branch-0.5 [6]. The overall architecture of Sparkpp in shown in Figure 1.

3.1 Resilient Distributed Dataset

We leverage the same abstraction of Resilient Distributed Dataset (RDD) [7] as Spark. We currently implement a subset of operations including lazy transformations including

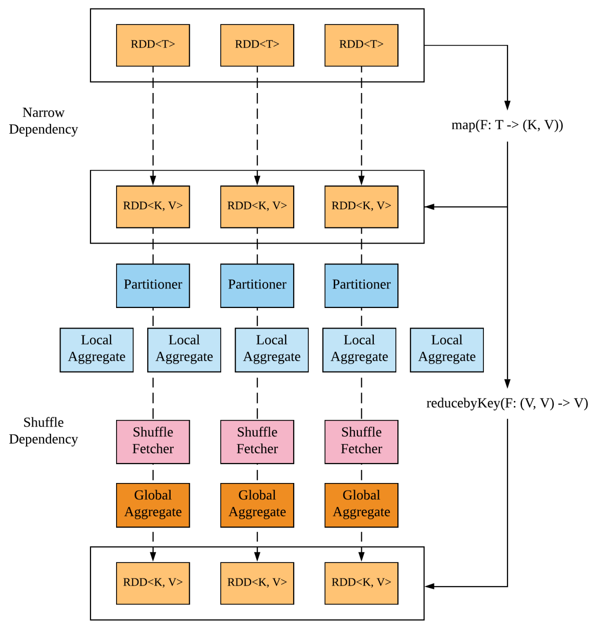

Figure 2. The illustration of RDD operations. The dynamic type of three RDDs might be `ParallelCollection<T>`, `PairRDD<K, V>` and `ShuffledRDD<K, V, V>`.

`map`, `flatMap`, `groupByKey` and `reduceByKey`, and eager actions such as `reduce`, `count` and `collect`. Each RDD carries dependencies and computing requirements and implements virtual methods to correctly compute results and return iterators. The idea of RDD operations is shown in Figure 2. The SparkContext generates the initial data RDD (`ParallelCollection<T>`) and transformations generate different types of RDDs with dependencies and resources. The transformations are local operations which are automatically fused. Finally the actions invoke the DAGSchduler methods to run jobs distributedly.

3.2 SparkContext

Our Sparkpp currently only implements distributed execution mode. SparkContext collects the input arguments and selects either master or slave mode to start. Slave mode SparkContext starts a thread pool of executors waiting for remote tasks, executing, and sending back result bytes. Master mode SparkContext sets up initial state and continues user functions. In addition, SparkContext provides interfaces for interacting with DAGScheduler (`runJob`) and generating new RDDs (`parallelize`).

3.3 CacheTracker

The CacheTracker server runs in master node and collects cache addition and drop events from all nodes. The RDDs will be registered in CacheTracker by DAGScheduler. When computing, the RDDs will first try to fetch the computed results from the local cache, if failed, then do the real computation and add results to cache.

3.4 MapOutputTracker and ShuffleManager

The MapOutputTracker server tracks all the shuffle results positions. It maintains all server uris to access remote shuffle files. The ShuffleManagers exist in each node serving as HTTP servers to response requests for local shuffle files. When a shuffling task is executing, it uses ShuffleFetcher to fetch remote shuffle files and do final aggregation work.

3.5 Lineage Tracking

RDDs can specify their dependencies and DAGScheduler collects the dependencies to build the directed acyclic graph. There are two kinds of dependencies: NarrowDependency represents the one to one RDD correspondence, which means computations can be fused into a single computation; ShuffleDependency represents the all to all multiplexing, which requires extra partitioner, aggregator and fetcher to partition, aggregate and fetch on disk data splits.

4 Details

4.1 Network Layer

We currently use Boost.Asio [8] as our TCP connection provider and Boost.Beast [9] to implement our HTTP Shuffle-Manager server. Currently our TCP and HTTP connections are both synchronous. The socket acceptors keep listening on specific ports for connections. We believe it's easy to convert current synchronous fashion HTTP servers into asynchronous ones, and improves the performance of fetching shuffled files.

4.2 Serialization

Data serialization To serialize our data, we incorporate two level of serialization frameworks. Sparkpp leverages Cap'n Proto [10] to represent variable-length messages and takes advantage of Boost.Serialization [11] to build the serialization protocol. In the future, it is possible to use a single layer for bytes representation of objects to reduce copies between different system layers.

Function serialization The key challenge for native distributed computing frameworks is how to spread user defined closures and execute closures in remote hosts.

- **RPC** Traditional RPC systems, like Apache Thrift [12], restrict the interfaces between clients and servers. They don't support user defined function except for passing shared address space function pointers.
- **Bytecode or IR** Managed languages like Java, C#, serialize functions as bytecodes executed by runtime interpreters. They usually leverage Just-In-Time compilers (tracing, partial evaluation) to speed up bytecode execution. We expect this method will involve a large

exectuion overhead and we don't want to lose performance.

- **Loading dynamic library** Like kernel modules or BSD packet filter [13], by asking users to write dynamically linked libraries and dynamically loading them in executors, we can serialization functions to remote servers. However, this method requires type registering and type information serializations. Otherwise the dynamical libraries can only receive untyped bytes, which can produce lots of unintended errors. Moreover, this method disables link time compilation optimization (LTO) [14] across compilation units.
- **Shared binary with serialized virtual table** We choose to send same binary files to different nodes in a MPI [5] fashion, and serialize functions by passing implementation-defined virtual table bytes. The type checking is done as compile time by compilers and LTO is enabled. Sparkpp current requires disabling Address space layout randomization (ASLR) to make virtual table addresses consistent. A more compatible solution is to serialize virtual table offsets defined in Application binary interfaces (ABIs). With further compiler support, we can further transform virtual callings to looking up the global function table, therefore eliminating the virtual dispatching overhead.

4.3 Native memory management

Thanks to modern C++ `unique_ptr<T>` and `shared_ptr<T>`, Sparkpp currently manages memory automatically in a Resource-Acquisition-Is-Initialization (RAII) fashion. The executors directly revive serialized objects on user-space network buffers to avoid extra copies, and releases dynamically allocated resources after executions complete. We currently leverage *tcmalloc* in *gperftools* [15] as our memory allocator since it's specially designed for multi-threaded pooling objects. There are several options (E.g. jemalloc [16], mimalloc [17]) to choose from, and we havn't focus on fine-tuned tcmalloc parameters yet.

Since we can directly manipulate data storage (E.g. specialized `ParallelCollection<T>`), we can easily do cache-aware optimizations and columnar storage transformations in the future.

4.4 Computation

Sparkpp current employs Boost.Asio thread pools for computing tasks and we set thread pool capacity to `std::thread::hardware_concurrency` to fully utilize the computation resources. We also introduce OpenMP [18] for improving performance of user-defined functions. Potentially, we can introduce fine-tuned work-sharing/work-stealing thread pools (E.g. Cilk [19, 20])

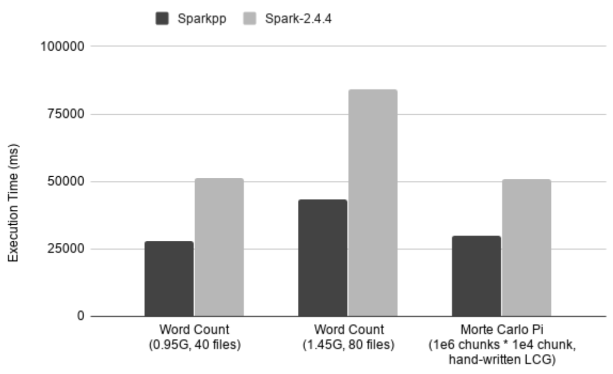

Figure 3. Experimental result for performance tests comparing Sparkpp and Spark-2.4.4

5 Evaluation

5.1 Experiment Setup

We conduct our experiments with 3 (1 master, 2 slaves) Amazon EC2 t3a.large instances. Each instance is equipped with 2 2.5Ghz CPU (4 logical threads available), 8 GB memory and 8 GB disk. Note that for scalability test, we use up to 4 nodes.

The word count test dataset is downloaded from Pets and Supplies reviews from Amazon review data 2018 version [21].

Sparkpp is compiled with compiler flags *-O3 -DNDEBUG -march=native -mtune=native -fwhole-program -flto -Wl,–no-as-needed -l...* and Spark2.4.4 is started in standalone mode with default flags.

5.2 Performance

Due to project deadline and resource limit, we only implement two use cases (Word count and Morte Carlo Pi) and three experiments. The sample code snippets are shown in 4 and 5. We compare the performance between Sparkpp and Spark-2.4.4 standalone mode, which is highly optimized compared to Spark-0.5. We run all test cases multiple times to avoid impacts of linux file system cache and not fully optimized by JIT bytecodes. The result is shown in Figure 3. We notice that in all test cases, the execution time is reduced by 40% to 50%. Meanwhile, the roughly memory usage for Sparkpp and Spark-2.4.4 for 40 files task is 1.13G and 2.87G.

We also profile the executor usage of the Word Count test case using google-gprof [15]. It turns that the bottleneck is the hashing procedure in `std::unordered_map<string, int>`. By incorporating specialized version of hash map in aggregation (E.g. `tsl::array_map` [22] and `google::dense_hash_map` [23]), we believe Sparkpp can reduce a large amount of execution time on text processing tasks. The profiling report is attached in Appendix 3.

```scala
import scala.io.Source
val files = 0 until 80
spark.time(sc
      .parallelize(files, 80)
      .flatMap(idx =>
        Source.fromFile(f"path/input_$idx")
              .getLines
              .flatMap(l => l.split(' ')))
      .map(s => (s, 1))
      .reduceByKey(_ + _, 8)
      .collect())
```

```cpp
// some includes & initializations
// init by std::iota
vector files = {0, ..., 79};
auto t_start = steady_clock::now();
auto result =
sc.parallelize(files, 80);
  .flatMap([](size_t v) noexcept {
    auto path = fmt::format("path/input_{}", v);
    std::ifstream ifs{path}, vector<vector<string>> words;
    for (string line; getline(ifs, line);) {
        vector<string> w;
        boost::algorithm::split(w, move(line),
            [](const char c) { return c == ' '; });
        words.push_back(move(w));
    }
    return flatten(words);
})
 .mapPair([](string&& w) noexcept {
    return make_pair(move(w), 1); });
 .reduceByKey([](int a, int b) noexcept {
    return a + b; }, 8)
 .collect();
auto t_end = steady_clock::now();
```

Figure 4. Word count code for Spark and Sparkpp. Both versions read file lazily (getLines and std::getline). Note that Sparkpp actually requires separate rdd declarations for chained calling (due to lack of enabled_shared_from_this in current version of Sparkpp).

```scala
val chunks = 1e6.toLong
val chunkSize = 1e4.toLong
val values = 0L until chunks
def ran(i: Long): Long = {
    var count = 0L
    var prev = i
    for (j <- 0 to 1e4.toInt) {
        prev =
    (prev * 998244353L + 19260817L) % 134456;
        val x: Double = prev / 67228.0 - 1;
        prev =
    (prev * 998244353L + 19260817L) % 134456;
        val y: Double = prev / 67228.0 - 1;
        if (x * x + y * y < 1) {
            count += 1;
        }
    }
    count
}
spark.time(sc.parallelize(values, 4)
    .map(i => ran(i))
    .reduce((a, b) => a + b))
```

```cpp
// some includes & initializations
// init by std::iota
vector chunks = {0ll, ..., 1e6ll};
auto t_start = steady_clock::now();
auto result =
sc.parallelize(chunks, 4);
  .map([](long long n) noexcept {
    unsigned long long count = 0;
    for (auto i = 0; i < chunkSize; ++i) {
        n = (n * 998244353ll + 19260817ll) % 134456;
        double x = n / 67228.0 - 1;
        n = (n * 998244353ll + 19260817ll) % 134456;
        double y = n / 67228.0 - 1;
        if (x * x + y * y < 1) { ++count; }
    }
    return count;
})
 .reduce([](ull n, ull m) noexcept { return n + m; });
auto t_end = steady_clock::now();
```

Figure 5. Morte carlo Pi code for Spark and Sparkpp.

```
// some includes & initializations
// init by std::iota
vector chunks = {0ll, ..., nll};
auto t_start = steady_clock::now();
auto result =
sc.parallelize(chunks, partitions);
  .map([](long long n) noexcept {
    unsigned long long count = 0;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis{
        -1.0, 1.0};
    // for OpenMP tests
    #pragma omp parallel for reduction(+:count) \
        default(none) private(dis, gen)
    for (auto i = 0; i < chunkSize; ++i) {
        double x = dis(gen);
        double y = dis(gen);
        if (x * x + y * y < 1) {
            ++count;
        }
    }
    return count;
})
 .reduce([](ull n, ull m) noexcept {
    return n + m; });
auto t_end = steady_clock::now();
```

Figure 6. Morte Carlo Pi with slow random function

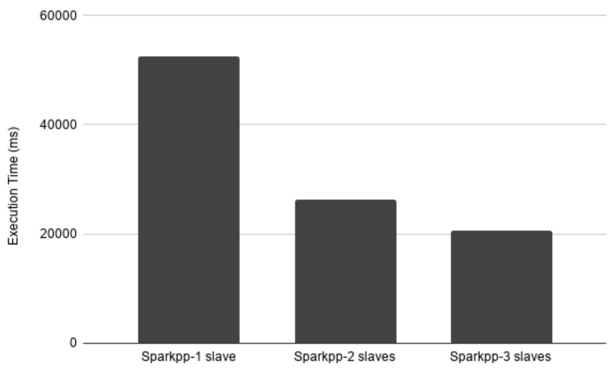

Figure 7. Scalability test for Sparkpp

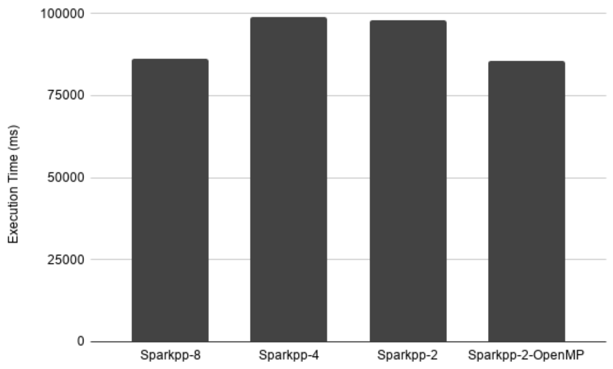

Figure 8. OpenMP test for Sparkpp

5.3 Scalability

Due to project deadline and resource limit, we only conduct scalability test on 1, 2, and 3 slave nodes to see if Sparkpp fully utilizes the nodes. The test case is Morte Carlo Pi + slow random function with 24 chunk and 10^8 computations per chunk. The test code is shown in 6. The result is shown in Figure 7. We notice in three slave cases, the performance is higher than expected. We suppose it's due to imbalanced workloads to spread 24 partitions to 3 nodes.

5.4 OpenMP

The theoretical parallelism our two slaves can achieve is 8 tasks (2 (slaves) × 2 (cores) × 2 (hyperthreading). We test the performance under different partition numbers and with OpenMP support for Morte Carlo Pi + slow random function with 8 chunks and 10^9 computatons per chunk. The code is shown in 6 and the result is illustrated in 8. It shows that though computatiation with 2 or 4 partitions can not fully utilize the computation units, by applying OpenMP optimization, poorly partitioned tasks can reach the same performance as correctly partitioned tasks.

6 Discussion

6.1 Introducing C++ ecosystem

We believe Sparkpp can serve as a bridge importing C++ utilities into data intensive applications. Without extra Foreign Function Interface overhead (E.g. JNI, C++/CLI), users can directly write or transform their OpenMP, MPI (requiring customized process manager instead of *mpirun*) [5], CUDA [24] and Charm++ [25] code to slave nodes.

Meanwhile, C++ compilers are highly optimized, including GNU C++, Clang with LLVM [26]. It's easy to do interprocedural optimization (including link time optimizations, whole program optimizations) and profile-guided optimization (E.g. *-fprofile-** in GCC, AutoFDO [27]).

Finally, getting rid of managed runtime and memory gives Sparkpp flexibility to manipulate memory and instructions as a low level. People can use CPU intrinsics to speed up computations and columnar or cache-friendly alignment of objects to reduce memory footprint and increase performance.

6.2 Drawbacks

Though it sounds glorious, Sparkpp still has several inherent flaws by its shared binary with serialized virtual table design.

- **Can't handle heterogeneous commodity hardware** Sparkpp requires all nodes to be machine code compatible (E.g. SSE, AVX versions) and if dynamically compiled, requires all dynamic library having the same version or ABI-compatible versions. Meanwhile, since virtual table is implemented-defined by compilers and guarded by ABIs (E.g. Itanium ABI [28]), Sparkpp also requires system C runtime libraries (for example, glibc for `dynamic_cast`, RTTI) to be compatible. These requirements are unrealistic for large clusters, which makes it fall back to byte code interpreting mechanism, which is nearly same as rewriting a JVM.
- **Long compiling time** C++ is infamous for its large compiling time, especially with head-only templates. Unforuntately, due to generic nature of RDD operations and user-defined functions, it's evitable to maintain Sparkpp as a nearly header-only library, What's worse, the shared binary way requires every application to be compiled once even for a single correction. Things can get worse if statically linking external libraries. Distributed shared compilation cache (E.g. sccache [29]) could relieve this problem if lots of users share a few applications.

6.3 Spark trending

In latest version of Apache Spark, it starts to support a different set of API: DataFrame/DataSet and new execution engine [30]. Due to project deadline limit, we haven't test the new data frame APIs performance for our tests. It exposes more Spark execution to native, bare metal execution by JNI callings and native intrinsics in JVM (E.g. `sun.misc.Unsafe`). We consider these effort as providing a intermediate layer bridging native machine code and JVM execution, namely a mixture of Sparkpp and Scala Spark.

7 Conclusion

Apache Spark is a high performance, well-known and widely-used distributed data parallel framework. In this report, we implementing a native version of Apache Spark, Sparkpp. We also conduct several tests comparing to lateest Spark implementation in Scala. The result shows that we are able to improve the performance and reduce memory usage of distributed applications.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[2] native_spark. https://github.com/rajasekarv/native_spark.

[3] Raja Sekar. Fastspark: A new fast native implementation of spark from scratch. https://medium.com/@rajasekar3eg/fastspark-a-new-fast-native-implementation-of-spark-from-scratch-368373a29a5c.

[4] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. *CoRR*, abs/1608.05634, 2016.

[5] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[6] Apache spark. https://github.com/apache/spark.

[7] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[8] Boost.asio. https://github.com/boostorg/Asio.

[9] Boost.beast http and websocket built on boost.asio in c++11. https://github.com/boostorg/beast.

[10] Cap'n proto. https://github.com/capnproto/capnproto.

[11] Boost.serialization. https://github.com/boostorg/serialization.

[12] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

[13] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[14] T. Glek and Jan Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, abs/1010.2196, 2010.

[15] gperftools. https://github.com/gperftools/gperftools.

[16] jemalloc. https://github.com/jemalloc/jemalloc.

[17] mimalloc. https://github.com/microsoft/mimalloc.

[18] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[20] A. D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science Engineering*, 15(2):66–71, March 2013.

[21] Jianmo Ni. Amazon review data (2018). https://nijianmo.github.io/amazon/index.html.

[22] Array-hash. https://github.com/Tessil/array-hash.

[23] sparsehash-c11. https://github.com/sparsehash/sparsehash-c11.

[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[25] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 12–23, New York, NY, USA, 2016. ACM.

[28] Itanium c++ abi. https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[29] Shared compilation cache. https://github.com/mozilla/sccache.

[30] Project tungsten: Bringing apache spark closer to bare metal, Feb 2018.

8 Appendix

8.1 Word Count executor profiling report

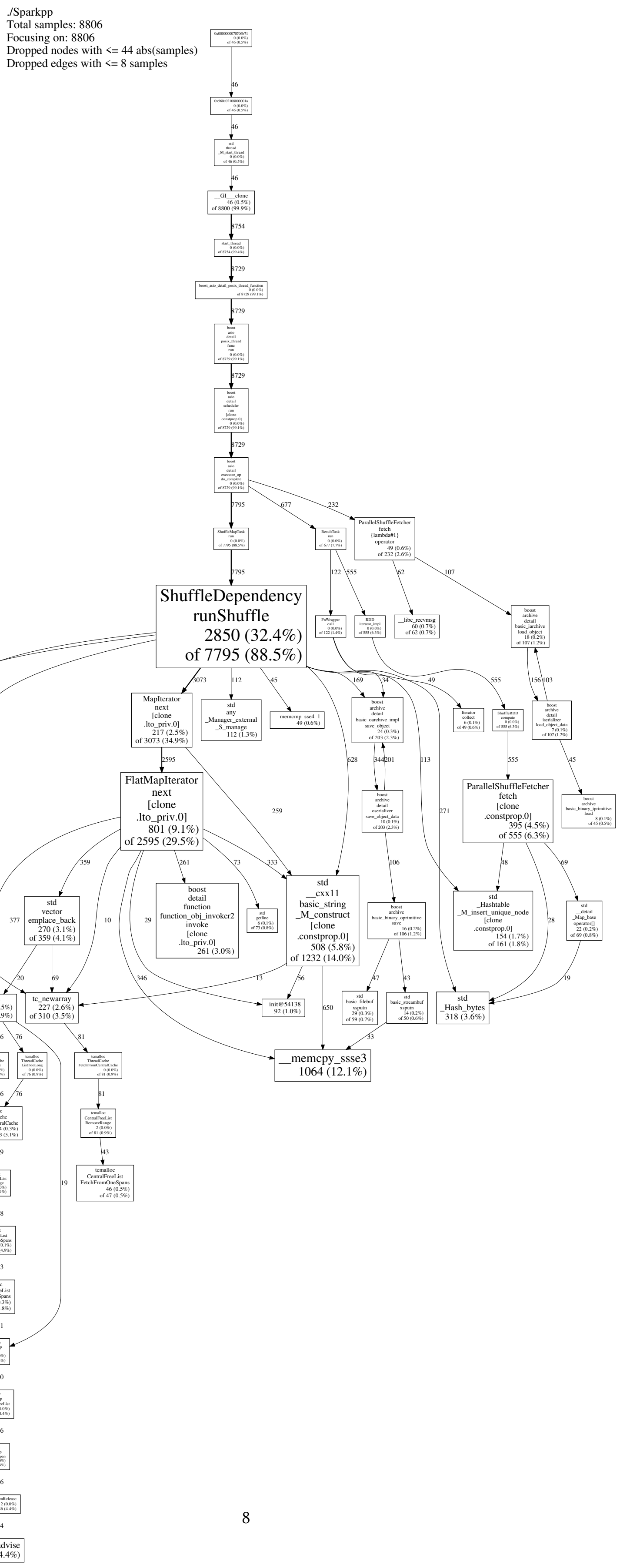
./Sparkpp
Total samples: 8806
Focusing on: 8806
Dropped nodes with <= 44 abs(samples)
Dropped edges with <= 8 samples
__GI___clone 46 (0.5%) of 8800 (99.9%)
ShuffleDependency runShuffle 2850 (32.4%) of 7795 (88.5%)
ParallelShuffleFetcher fetch [lambda#1] operator 49 (0.6%) of 232 (2.6%)
__libc_recvmsg 60 (0.7%) of 62 (0.7%)
std _Hashtable ~_Hashtable 80 (0.9%) of 137 (1.6%)
MapIterator next [clone .lto_priv.0] 217 (2.5%) of 3073 (34.9%)
std any _Manager_external _S_manage 112 (1.3%)
__memcmp_sse4_1 49 (0.6%)
boost archive detail basic_oarchive_impl save_object 24 (0.3%) of 203 (2.3%)
ParallelShuffleFetcher fetch [clone .constprop.0] 395 (4.5%) of 555 (6.3%)
FlatMapIterator next [clone .lto_priv.0] 801 (9.1%) of 2595 (29.5%)
std vector emplace_back 270 (3.1%) of 359 (4.1%)
boost detail function function_obj_invoker2 invoke [clone .lto_priv.0] 261 (3.0%)
std __cxx11 basic_string _M_construct [clone .constprop.0] 508 (5.8%) of 1232 (14.0%)
std _Hashtable _M_insert_unique_node [clone .constprop.0] 154 (1.7%) of 161 (1.8%)
cfree 133 (1.5%) of 604 (6.9%)
tc_newarray 227 (2.6%) of 310 (3.5%)
_init@54138 92 (1.0%)
std _Hash_bytes 318 (3.6%)
__memcpy_ssse3 1064 (12.1%)
__GI_madvise 384 (4.4%)

Figure 9. Profiler report by google-gprof