Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++

Timo Bingmann*, Michael Axtmann*, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen,
Alexander Noe, Sebastian Schlag*, Matthias Stumpp, Tobias Sturm, and Peter Sanders*
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
**Emails: {firstname.lastname}@kit.edu*

***Abstract*—We present the design and a first performance evaluation of Thrill – a prototype of a general purpose big data processing framework with a convenient data-flow style programming interface. Thrill is somewhat similar to Apache Spark and Apache Flink with at least two main differences. First, Thrill is based on C++ which enables performance advantages due to direct native code compilation, a more cache-friendly memory layout, and explicit memory management. In particular, Thrill uses template meta-programming to compile chains of subsequent local operations into a single binary routine without intermediate buffering and with minimal indirections. Second, Thrill uses arrays rather than multisets as its primary data structure which enables additional operations like sorting, prefix sums, window scans, or combining corresponding fields of several arrays (zipping).**

We compare Thrill with Apache Spark and Apache Flink using five kernels from the HiBench suite. Thrill is consistently faster and often several times faster than the other frameworks. At the same time, the source codes have a similar level of simplicity and abstraction.

***Keywords*-C++; big data tool; distributed data processing;**

I. Introduction

In this paper we present Thrill, a new open-source C++ framework for algorithmic distributed batch data processing.

The need for parallel and distributed algorithms cannot be ignored anymore, since individual processor cores' clock speeds have stagnated in recent years. At the same time, we have experienced an explosion in data volume so that scalable distributed data analysis has become a bottleneck in an ever-increasing range of applications. With Thrill we want to make a step at bridging the gap between two traditional scenarios of "Big Data" processing.

On the one hand, in academia and high-performance computing (HPC), distributed algorithms are often handcrafted in C/C++ and use MPI for explicit communication. This achieves high efficiency at the price of difficult implementation. On the other hand, global players in the software industry created their own ecosystem to cope with their data analysis needs. Google popularized the MapReduce [**?**] model in 2004 and described their in-house implementation. Apache Hadoop and more recently Apache Spark [**?**] and Apache Flink [**?**] have gained attention as open-source Scala/Java-based solutions for heavy duty data processing. These frameworks provide a simple programming interface and promise *automatic* work parallelization and scheduling, *automatic* data distribution, and *automatic* fault tolerance. While most benchmarks highlight the scalability of these frameworks, the bottom line efficiency has been shown to be lacking [**?**], surprisingly with the CPU often being the bottleneck [**?**].

Thrill's approach to bridging this gap is a library of *scalable algorithmic primitives* like *Map*, *ReduceByKey*, *Sort*, and *Window*, which can be combined efficiently to construct larger complex algorithms using pipelined data-flow style programming. Thrill is written in modern C++14 from the ground up, has minimal external dependencies, and compiles cross-platform on Linux, Mac OS, and Windows. By using C++, Thrill is able to exploit compile-time optimization, template meta-programming, and explicit memory management. Thrill enables efficient processing of fixed-length items like single characters or fixed-dimensional vectors without object overhead due to the zero overhead abstractions of C++. It treats data types of operations as opaque and utilizes template programming to instantiate operations with user-defined functions (UDFs). For example, the comparison function of the sorting operation is compiled into the actual internal sorting and merging algorithms (similar to *std::sort*). At the same time, Thrill makes no attempts to optimize the execution order of operations, as this would require introspection into the data and how UDFs manipulate it.

Thrill programs run in a collective bulk-synchronous manner similar to most MPI programs. Thrill primarily focuses on fast in-memory computation, but transparently uses external memory when needed. The functional programming style used by Thrill enables easy parallelization, which also works remarkably well for shared memory parallelism. Hence, due to the restriction to scalable primitives, Thrill programs run on a wide range of homogeneous parallel systems.

By using C++, Thrill aims for high performance distributed algorithms. JVM-based frameworks are often slow due to the overhead of the interpreted bytecode, even though just-in-time (JIT) compilation has leveled the field somewhat. Nevertheless, due to object indirections and garbage collection, Java/Scala must remain less cache-efficient. While efficient CPU usage should be a matter of

course, especially when processing massive amounts of data, the ultimate bottleneck for scalable distributed application is the (bisection) bandwidth of the network. But by using more tuned implementations, more CPU time is left for compression, deduplication [**?**], and other algorithms to reduce communication. Nevertheless, in smaller networks the CPU is often the bottleneck [**?**], and for most applications a small cluster is sufficient.

A consequence of using C++ is that memory management has to be done explicitly. While this is desirable for more predictable and higher performance than garbage collected memory, it does make programming more difficult. However, with modern C++11 this has been considerably alleviated, and Thrill uses reference counting extensively outside of inner loops.

While scalable algorithms promise eventually higher performance with more hardware, the performance hit going from parallel shared memory to a distributed scenario is large. This is due to the communication latency and bandwidth bottlenecks. This network overhead and the additional management overheads of big data frameworks often make speedups attainable only with unjustifiable hardware costs [**?**]. Thrill cannot claim zero overhead, as network costs are unavoidable. But by overlapping computation and communication, and by employing binary optimized machine code, we keep the overhead small.

Thrill is open-source under the BSD 2-clause license and available as a community project on GitHub[1]. It currently has more than 55 K lines of C++ code and approximately a dozen developers have contributed.

Overview: The rest of this section introduces related work with an emphasis on Spark and Flink. Section II discusses the design of Thrill, in particular its API and the rationale behind the chosen concept. We present a complete WordCount example in Section II-B, followed by an overview of the current portfolio of operations. Details of their implementation and of pipelining are discussed in Section III. In Section IV, results of an experimental comparison of Thrill, Spark, and Flink based on six micro benchmarks including PageRank and KMeans are shown. Section V concludes with an outlook on future work.

Our Contributions: Thrill demonstrates that with the advent of C++11 lambda-expressions, it has become feasible to use C++ for big data processing using an abstract and convenient API comparable to currently popular frameworks like Spark or Flink. This not only harvests the usual performance advantages of C++, but allows us moreover to transparently compile sequences of local operations into a single binary code via sophisticated template meta-programming. By using arrays as the primary data type, we enable additional basic operations that have to be emulated by more complicated and more costly operations in traditional multiset-based systems. Our experimental evaluation demonstrates that even the current prototypical implementation already offers a considerable performance advantage over Spark and Flink.

[1] http://github.com/thrill/thrill

A. Related Work

Due to the importance and hype of the "Big Data" topic, a myriad of distributed data processing frameworks have been proposed in recent years [**?**]. These cover many different aspects of this challenge like data warehousing and batch processing, stream aggregation, interactive queries, and specialized graph and machine learning frameworks.

In 2004, Google established the MapReduce paradigm [**?**] as an easy-to-use interface for scalable data analysis. Their paper spawned a whole research area on how to express distributed algorithms using just *map* and *reduce* in as few rounds as possible. Soon, Apache Hadoop was created as an open-source MapReduce framework written in Java for commodity hardware clusters. Most notable from this collection of programs was the Hadoop distributed file system (HDFS) [**?**], which is key for fault tolerant data management for MapReduce. Subsequently, a large body of academic work was done optimizing various aspects of Hadoop like scheduling and data shuffling [**?**].

MapReduce and Hadoop are very successful due to their simple programming interface, which at the same time is a severe limitation. For example, iterative computations are reported to be very slow due to the high number of MapReduce rounds, each of which may need a complete data exchange and round-trip to disks. More recent frameworks such as Apache Spark and Apache Flink offer a more general interface to increase usability and performance.

Apache Spark operates on an abstraction called *resilient distributed datasets* (RDDs) [**?**]. This abstraction provides the user with an easy-to-use interface which consists of a number of deterministic coarse-grained operations. Spark can maintain already computed RDDs in main memory to be reusable by future operations, in order to speed-up iterative computations [**?**].

In more recent versions, Spark added two more APIs: DataFrames [**?**] and Datasets. Both offer domain specific languages for higher level declarative programming similar to SQL, which allows Spark to optimize the query execution plan. Even further, it enables Spark to generate optimized query bytecode online, aside of the original Scala/Java program. The DataFrame engine is built on top of the original RDD processing interface.

Apache Flink originated from the Stratosphere research project [**?**] and is progressing from an academic project to industry. While Flink shares many ideas with Spark such as the master-worker model, lazy operations, and iterative computations, it tightly integrates concepts known from parallel database systems. Flink's core interface is a domain-specific declarative language. Furthermore, Flink's focus has

turned to streaming rather than batch processing.

The interfaces of Spark and Flink differ in some very important ways. Flink's optimizer requires introspection into the components of data objects and how the UDFs operate on them. This requires many Scala/Java annotations to the UDFs and incurs an indirection for access to the values of components. In contrast to Spark's RDD interface, where users can make use of *host language control-flow*, Flink provides custom iteration operations. Hence, Flink programs are in this respect more similar to declarative SQL statements than to an imperative language. The newer DataFrame and Dataset interfaces introduce similar concepts to Spark, but extend them further with a custom code generation engine. At its core, Spark is an in-memory batch engine that executes streaming jobs as a series of mini-batches. In contrast, Flink is based on a pipelined execution engine used in database systems, allowing Flink to process streaming operations in a pipelined way with lower latency than in the micro-batch model. In addition, Flink supports external memory algorithms whereas Spark is mainly an in-memory system with spilling to external memory.

Overall the JVM is currently the dominant platform for open-source big data frameworks. This is understandable from the point of view of programmer productivity but surprising when considering that C++ is the predominant language for performance critical systems – and big data processing is inherently performance critical. Spark [**?**] (with Project Tungsten) and Flink (with MemorySegments) therefore put great efforts into overcoming performance penalties of the JVM, for example by using explicit *Unsafe* memory operations and generating optimized bytecode to avoid object overhead and garbage collection. With Thrill we present a C++ framework as an alternative that does not incur these overheads in the first place.

II. Design of Thrill

Thrill programs are written in C++ and compile into binary programs. The execution model of this binary code is similar to MPI programs: one identical program is run collectively on h machines. Thrill currently expects all machines to have nearly identical hardware, since it balances work and data equally between the machines. The binary program is started simultaneously on all machines, and connects to the others via a network protocol. Thrill currently supports TCP sockets and MPI as network backends. The startup procedures depend on the specific backend and cluster environment.

Each machine is called a *host*, and each work thread on a host is called a *worker*. Currently, Thrill requires all hosts to have the same number of cores c, hence, in total there are $p = h \cdot c$ worker threads. Additionally, each host has one thread for network/data handling and one for asynchronous disk I/O. Each of the h hosts have $h-1$ reliable network connections to the other hosts, and the hosts and workers are enumerated $0 \ldots h-1$ and $0 \ldots p-1$. Thrill does not have a designated master or driver host, as all communication and computation is done collectively.

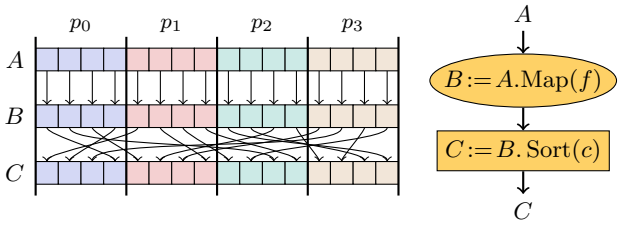

Figure 1. Distribution of a DIA between processors (left) and a data-flow graph (right)

Thrill currently provides no fault tolerance. While our data-flow API permits smooth integration of fault tolerance using asynchronous checkpoints [**?**], [**?**], the execution model of exactly h machines may have to be changed.

A. Distributed Immutable Arrays

The central concept in Thrill's high-level data-flow API is the *distributed immutable array* (DIA). A DIA is an array of items which is distributed over the cluster in some way. No direct array access is permitted. Instead, the programmer can apply so-called *DIA operations* to the array as a whole. These operations are a set of *scalable primitives*, listed in Table I, which can be composed into complex distributed algorithms. DIA operations can create DIAs by reading files, transform existing DIAs by applying user functions, or calculate scalar values collectively, used to determine the further program control flow. In a Thrill program, these operations are used to lazily construct a DIA *data-flow* graph in C++ (see Figure 1). The data-flow graph is only executed when an *action* operation is encountered. How DIA items are actually stored and in what way the operations are executed on the distributed system remains transparent to the user.

In the current prototype of Thrill, the array is usually distributed evenly between the p workers in order. DIAs can contain any C++ data type, provided serialization methods are available (more in Section III-B). Thrill contains built-in serialization methods for all primitive types, and many STL types; only custom non-trivial classes require additional methods. Each DIA operation in Table I is implemented as a C++ template class, which can be instantiated with appropriate UDFs.

B. Example: WordCount

We now present a complete code of the popular WordCount benchmark in Algorithm 2 to demonstrate how easy it is to program in Thrill. The program counts the number of occurrences of each unique word in a text. In Thrill, WordCount including file I/O consists of five DIA operations.

ReadLines (line 4) and *WriteLines* (line 22) are used to read the text and write the result from/to the file system. Thrill currently uses standard POSIX filesystem methods

```
1 void WordCount(thrill::Context& ctx,
2   std::string input, std::string output) {
3   using Pair = std::pair<std::string, size_t>;
4   auto word_pairs = ReadLines(ctx, input)
5   .FlatMap<Pair>(
6     // flatmap lambda: split and emit each word
7     [](const std::string& line, auto emit) {
8       Split(line, ' ', [&](std::string_view sv) {
9         emit(Pair(sv.to_string(), 1));
10      });
11    });
12  word_pairs.ReduceByKey(
13    // key extractor: the word string
14    [](const Pair& p) { return p.first; },
15    // commutative reduction: add counters
16    [](const Pair& a, const Pair& b) {
17      return Pair(a.first, a.second + b.second);
18    })
19  .Map([](const Pair& p) {
20    return p.first + ": "
21           + std::to_string(p.second); })
22  .WriteLines(output);
23 }
```

Figure 2. Complete WordCount Example in Thrill

to read and write to disk, and it requires a distributed parallel file system such as NFS, Lustre, or Ceph to provide a common view to all compute hosts. ReadLines takes a *thrill::Context* object, which is only required for source DIA operations, and a set of files. The result of ReadLines is a *DIA⟨std::string⟩*, which contains each line of the files as an item. The set of files is ordered lexicographically and the set of lines is partitioned equally among the workers.

However, this DIA is not assigned to a variable name. Instead, we immediately append a *FlatMap* operation (line 5) which splits each text line into words and emits one *std::pair⟨std::string,size_t⟩* (aliased as *Pair*) containing $(word, 1)$ per word. In the example, we use a custom *Split* function and *std::string_view* to reference characters in the text line, and copy them into word strings (lines 7–11). The *emit auto* parameter of the *FlatMap* lambda function (line 7) enables Thrill to pipeline the *FlatMap* with the following *ReduceByKey* operation. Details on pipelining are discussed in Section III-A. The result of *FlatMap* is a *DIA⟨Pair⟩*, which is assigned to the variable *word_pairs*. Note that the keyword *auto* makes C++ infer the appropriate type for *word_pairs* automatically.

The operation *ReduceByKey* is then used to reduce $(word, 1)$ pairs by *word*. This DIA operation must be parameterized with a key extractor (take *word* out of the pair, line 14) and a reduction function (sum two pairs with the same key together, line 17). Thrill currently implements *ReduceByKey* using hash tables, as described in Section III-C. Notice that C++ will infer most types during instantiation of *ReduceByKey*, both input and output are implicit; only with *FlatMap* it is necessary to specify what type gets emitted.

The output of *ReduceByKey* is again a *DIA⟨Pair⟩*. We need to use a *Map* to transform the *Pairs* into printable strings (lines 19–21), which can then be written to disk using the *WriteLines* action. Again, the return type of the *Map* (*std::string*) is inferred automatically, and hence the result of the *Map* operation is implicitly a *DIA⟨std::string⟩*.

Notice that it is not obvious that the code in Algorithm 2 describes a parallel and distributed algorithm. It is the *implementation* of the DIA operations in the lazily built data-flow graph which perform the actual distributed execution. The code instructs the C++ compiler to instantiate and optimize these template classes with the UDFs provided. At runtime, objects of these template classes are procedurally created and evaluated when actions are encountered in the DIA data-flow graph.

C. Overview of DIA Operations

Table I gives an overview of the DIA-operations currently supported by Thrill. The immutability of a DIA enables functional-style data-flow programming. As DIA operations can depend on other DIAs as inputs, these form a directed acyclic graph (DAG), which is called the *DIA data-flow graph*. We denote DIA operations as vertices in this graph, and directed edges represent a dependency. Intuitively, one can picture a directed edge as the *values* of a DIA as they flow from one operation into the next.

We classify all DIA operations into four categories. *Source* operations have no incoming edges and generate a DIA from external sources like files, database queries, or simply by generating the integers $0 \ldots n-1$. Operations which have one or more incoming edges and return a DIA are classified further as *local* (LOps) and *distributed* operations (DOps). Examples of LOps are *Map* or *Filter*, which apply a function to every item of the DIA independently. LOps can be performed locally and in parallel, without any communication between workers. On the other hand, DOps such as *ReduceByKey* or *Sort* may require communication and a full data round-trip to disks.

The fourth category are *actions*, which do not return a DIA and hence have no outgoing edges. The DIA data-flow graph is built lazily, i.e. DIA operations are not immediately executed when created. Actions trigger evaluation of the graph and return a value to the user program. For example, writing a DIA to disk or calculating the sum of all values are actions. By inspecting the results of actions, a user program can determine the future program flow, e.g. to iterate a loop until a condition is met. Hence, control flow decisions are performed *collectively* in C++ with imperative loops or recursion (host language control-flow).

Initial DIAs can be generated with Thrill's *source* operations. *Generate* creates a DIA by mapping each index $[0, size)$ to an item using a generator function. *ReadLines* and *ReadBinary* read data from the file system and create a DIA with this data.

Thrill's *FlatMap* LOp corresponds to the *map* step in the MapReduce paradigm. Each item of the input DIA is

Table I
DIA OPERATIONS OF THRILL

Operation	User Defined Functions
Sources	
Generate$(n) : [0, \ldots, n-1]$	n : DIA size
Generate$(n, g) : [A]$	g : **unsigned** $\to A$
ReadLines() : files $\to$ [**string**]	
ReadBinary$\langle A \rangle$() : files $\to [A]$	A : data type
Local Operations (no communication)	
Map$(f) : [A] \to [B]$	$f : A \to B$
FlatMap$(f) : [A] \to [B]$	$f : A \to$ **list**(B)
Filter$(f) : [A] \to [A]$	$f : A \to$ **bool**
BernoulliSample$(p) : [A] \to [A]$	p : success probability
Union$() : [A] \times [A] \cdots \to [A]$	
Collapse$() : [A] \to [A]$	
Cache$() : [A] \to [A]$	
Distributed Operations (communication between workers)	
ReduceByKey(k, r) : **ReduceToIndex**(i, r, n) : $[A] \to [A]$ **GroupByKey**(k, g) : **GroupToIndex**(i, g, n) : $[A] \to [B]$	$k : A \to K$ $i : A \to [0, n)$ $r : A \times A \to A$ $g :$ **iterable**$(A) \to B$ n : result size
Sort$(c) : [A] \to [A]$	$c : A \times A \to$ **bool**
Merge$(c) : [A] \times [A] \cdots \to [A]$	$c : A \times A \to$ **bool**
Concat$() : [A] \times [A] \cdots \to [A]$	
PrefixSum$(s, i) : [A] \to [A]$	$s : A \times A \to A$ i : initial value
Zip$(z) : [A] \times [B] \cdots \to [C]$	$z : A \times B \cdots \to C$
ZipWithIndex$(z) : [A] \to [B]$	$z :$ **unsigned** $\times A \to B$
Window$(k, w) : [A] \to [B]$ **FlatWindow**$(k, f) : [A] \to [B]$	k : window size $w : A^k \to B$ $f : A^k \to$ **list**(B)
Actions	
Execute()	
Size$() : [A] \to$ **unsigned**	
AllGather$() : [A] \to$ **list**(A)	
Sum$(s, i) : [A] \to A$ **Min**$(i) : [A] \to A$ **Max**$(i) : [A] \to A$	$s : A \times A \to A$ i : initial value
WriteLines() : [**string**] $\to$ files **WriteBinary**() : $[A] \to$ files	

The notation $[A]$ is short for *DIA*$\langle A \rangle$, where A is an item type.

mapped to zero, one, or more output items by a function f. In C++ this is done by calling an *emit* function for each item, as shown in the WordCount example. All items are concatenated in order and form a new DIA. Special cases of *FlatMap* are *Map*, which maps each item to exactly one output, *Filter*, which selects a subset of the input DIA, and *BernoulliSample*, which samples each item independently with constant probability p. The LOp *Union* fuses two or more DIAs into one without regard for item order. In contrast, the DOp *Concat* keeps the order of the input DIAs and concatenates them, which requires communication.

Cache explicitly materializes the result of a DIA operation for later use. *Collapse* on the other hand folds a pipeline of functions, as described in more detail in Section III-C.

The *reduce* step from the MapReduce paradigm is represented by Thrill's *ReduceByKey* and *GroupByKey* DOps. In both operations, input items are grouped by a key. Keys are extracted from items using the *key extractor* function k, and then mapped to workers using a hash function h. In *ReduceByKey*, the associative reduction function r specifies how two items are combined into one. In *GroupByKey*, all items with a certain key are collected on one worker and processed by the group function g. When possible, *ReduceByKey* should be preferred as it allows local reduction and thus lowers communication volume and running time.

Both ReduceByKey and GroupByKey also offer a *ToIndex* variant, wherein each item of the input DIA is mapped by a function i to an index in the result DIA. The size of the resulting DIA must be given as n. Items which map to the same index are either reduced using an associative reduction function r, or processed by a group function g. Empty slots in the DIA are filled with a neutral item.

Sort sorts a DIA with a user-defined comparison function c and *Merge* merges multiple sorted DIAs, again using a user-defined comparison function c. *PrefixSum* uses an associative function s to compute the prefix sum (partial sum) for each item.

Zip combines two or more DIAs index-wise using a zip function z similar to functional programming languages. The function z is applied to all items with index i to deliver the new item at index i. The regular *Zip* function requires all DIAs to have equal length, but Thrill also provides variants which cut the DIAs to the shortest or pad them to the longest. *ZipWithIndex* zips each DIA item with its global index. While *ZipWithIndex* can be emulated using *Generate* and *Zip*, the combined variant requires less communication.

Window respects the ordering of a DIA and delivers all k consecutive items (a sliding window) to a function w which returns exactly one item. In the *FlatWindow* variant, the window function f can emit zero or more items. Thrill also provides specializations which delivers all disjoint windows of k consecutive items.

Sum is an action, which computes an associative function s over all items in a DIA and returns the result on every worker. By default *Sum* uses $+$. *Max* and *Min* are specializations of *Sum* with other operators. *Size* returns the number of items in a DIA and *AllGather* returns a whole DIA as *std::vector*$\langle T \rangle$ on each worker. *WriteLines* and *WriteBinary* write a DIA to the file system. *Execute* can be used to explicitly trigger evaluation of DIA operations.

Besides the actions which trigger evaluation, Thrill also provides *action futures*, *SumFuture*, *MinFuture*, *AllGatherFuture*, etc, which only insert an action vertex into the DIA data-flow graph, but *do not* trigger evaluation. Using action futures one can calculate multiple results (e.g. the minimum *and* maximum item) with just one data round trip.

The current set of scalable primitive DIA operations listed in Table I is definitely not final, and more distributed algorithmic primitives may be added in the future as necessary and prudent. In Section III-C we describe the implementations of some of the operations in more detail.

We also envision future work on how to accelerate scalable primitives, which can then be use as drop-in replacement to our current straight-forward implementations.

D. Why Arrays?

Thrill's DIA API is obviously similar to Spark and Flink's data-flow languages, which themselves are similar to many functional programming languages [**?**]. However, we explicitly define the items in DIAs to be ordered. This order may be arbitrary after operations like *ReduceByKey*, which hash items to indexes in the array, but they do have an order. Many of our operations like *PrefixSum*, *Sort*, *Merge*, *Zip*, and especially *Window* only make sense with an ordered data type.

Having an order on the distributed array opens up new opportunities in how to exploit this order in algorithms. Essentially, the order reintroduces the concept of *locality* into distributed data-flow programming. While one cannot access DIA items directly, such as in a imperative for loop over an array, one *can* iterate over them using a *Window* function *in parallel* with adjacent items as context. A common design pattern in Thrill programs is to use *Sort* or *ReduceToIndex* to bring items into a desired order, and then to process them using a *Window*. Furthermore, if the computation in a *Window* needs context from more than one DIA, these can be *Zip*-ped together first.

We are looking forward to future work on how this order paradigm can be exploited. Furthermore, extending Thrill beyond one-dimensional arrays to higher dimensional arrays, (sparse) matrices, or graphs is not only useful but also conceptually interesting since these data types have a more complex concept of locality.

III. Implementation of Thrill

A. Data-Flow Graph Implementation

Contrary to the picture of DIAs we have drawn for application programmers in the preceding sections, the distributed array of items usually does not exist explicitly. Instead, a DIA remains purely a conceptual data-flow between two concrete DIA operations. This data-flow abstraction allows us to apply an optimization called *pipelining* or *chaining*. Chaining in general describes the process of combining the logic of one or more functions into a single one (called *pipeline*). In Thrill we chain together all independently parallelizable local operations (*FlatMap*, *Map*, *Filter*, and *BernoulliSample*), and the first local computation steps of the next distributed DIA operation into one block of optimized binary code. Via this chaining, we reduce both the overhead of the data flow between them, as well as the total number of operations, and obviate the need to store intermediate explicit arrays. Additionally, we leverage the C++ compiler to combine the local computations *on the assembly level* with full optimization, thus reducing the number of indirections to a minimum, which additionally improves cache efficiency.

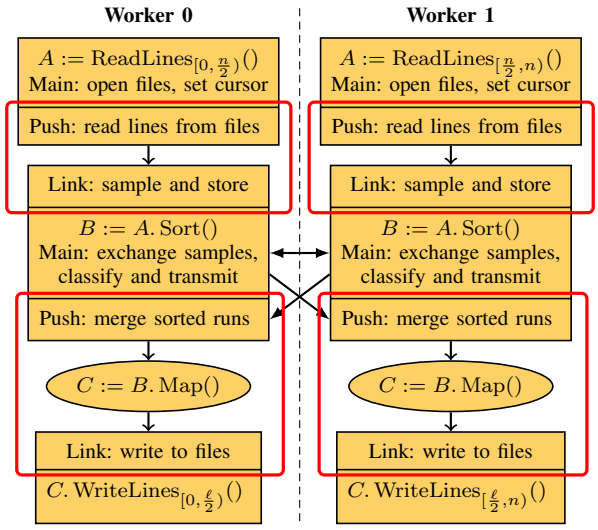

Figure 3. Subdivisions of DOps and chained Push, LOps, and Link parts

In essence, we combine all local computation of one bulk-synchronous parallel (BSP) superstep [**?**] using chaining into one block of assembly code.

To integrate the implementations of DIA operations into the pipelining framework we subdivide them into three parts: *Link*, *Main* and *Push* (see Figure 3 for an example). The Link part handles incoming items by performing some *finalizing local work* like storing or transmitting them. This process closes the pipeline and results in a single executable code block containing its logic. The Main part contains the actual DIA operation logic like sorting, synchronous communication, etc. And finally, the Push part represents the start of a new pipeline by *emitting* items for further processing. Depending on the type of a DIA operation, subdivisions can also be empty or trivial.

We explain these subdivisions using *PrefixSum* as an example. In the Link part, *PrefixSum* receives a stream of items from a preceding operation and stores them in sequence. While storing them, each worker keeps a local sum over all items. In the Main part, the workers perform a global synchronous exclusive prefix sum on the local sums to calculate the initial value of their items. This local initial value is then added to items while they are being read and Push-ed into the next operation.

Chaining also affects how data dependencies between DIA operations are represented in Thrill's data-flow graph. Due to pipelining of local operations into one assembly block, all LOp are fused with the succeeding DOp vertices. Hence only vertices representing distributed operations remain in the DAG. This optimized data-flow DAG corresponds to a set of BSP supersteps and their data dependencies, and is executed lazily when an action is encountered.

Execution is done by Thrill's *StageBuilder*, which performs a reverse breadth-first *stage* search in the optimized

DAG to determine which DIA operations need to be calculated. The gathered vertices are then executed in topological order such that their data dependencies are resolved prior to execution. Unnecessary recomputations are avoided by maintaining the state of each vertex, and DIA operations are automatically disposed via reference counting.

We implemented chaining and our execution model by making heavy use of C++ template programming. More precisely, we compose a pipeline by chaining together the underlying (lambda) functions using their static functor types. Since these types can be deduced by static analysis, chaining can take place during compile time, and hence chained operations can be optimized into single pipelined functions on the assembly code level. In the end all trivially-parallel local operation like *Map*, *FlatMap*, etc. introduce zero overhead during runtime, and are combined with the following DIA operation's *Link* part.

In Thrill we took pipelining of data processing one step further by enabling *consumption* of source DIA storage *while* pushing data to the next operation. DIA operations transform huge data sets, but a naive implementation would read all items from one DIA, push them all into the pipeline for processing, and then deallocate the data storage. Assuming the next operation also stores all items, this requires twice the amount of storage. However, with *consume* enabled, the preceding DIA operation's storage is deallocated while processing the items, hence the storage for all items is needed only once, plus a small overlapping buffer.

B. Data, Network, and I/O Layers

Below the convenient high-level DIA API of Thrill lie several software layers which do the actual data handling. DIA operations are C++ template classes which are chained together as described in Section III-A. These operations store and transmit the items using the *data*, *net*, and *io* layers.

Items have to be serialized to byte-sequences for transmission via the network or for storage on disk. Thrill contains a custom C++ serialization framework which aims to deliver high performance and low to zero overhead. This is possible because neither signatures nor versioning are needed. In general, fixed-length trivial items like integers and fixed-size numerical vectors are stored with zero overhead. Variable length items like strings and variable-length vectors are prepended with their length. Compound objects are stored as a sequence of their components.

DIA operations process a stream of items, which need to be transmitted or stored, and then read. Such a stream of items is serialized directly into the memory buffer of a *Block*, which is by default 2 MiB in size. Items in a Block are stored without separators or other per-item overhead. This is possible because Thrill's serialization methods correctly advance a cursor to the next item. Hence, currently only four integers are required as overhead per Block and zero per item. This efficient Block storage format is important for working with small items like plain integers or characters, but Thrill can also process large blobs spanning multiple Blocks.

A sequence of Blocks is called a *File*, even though it is usually stored in main memory. DIA operations read/write items sequentially to/from Files using template *BlockReader* and *BlockWriter* classes.

To transmit items to other workers, DIA operations have two choices. One is a set of efficient *synchronous* collective communication primitives similar to MPI, such as *AllReduce*, *Broadcast*, and *PrefixSum*. These utilize the same serialization framework and are mostly used for blocking communication of small data items, e.g. an integer AllReduce is often used to calculate the total number of items in a DIA.

The second choice are *Streams* for transmitting large amounts of items *asynchronously*. Streams enable bulk all-to-all communication between all workers. Thrill contains two subtypes of Streams which differ in the order items are received from other workers: *CatStream*s deliver items strictly in worker rank order, while *MixStream*s deliver items in the arbitrary order in which Blocks are received from the network. Besides transmitting items in Blocks using the BlockReader and BlockWriter classes, Streams can also scatter whole ranges of a File to other workers without an additional deep copy of the Block's data in the network layer. Items in Blocks scattered via Streams to workers on the same host are "transmitted" via reference counting and not deeply copied. All communication with workers on the same host is done via shared memory within the same process space.

All Blocks in a Thrill program are managed by the *BlockPool*. Blocks are reference counted and automatically deleted once they are no longer in any File or used by the network system. The BlockPool also keeps track of the total amount of memory used in Blocks. Once a user-defined limit is exceeded, the BlockPool asynchronously swaps out the least recently used Blocks to a local disk. To distinguish which Blocks may be evicted and which are being used by the data system, Blocks have to be *pinned* to access their data. Pins can be requested asynchronously to enable prefetching from external memory. However, all the complexity of pinning Blocks is hidden in the BlockReader/Writer such as to make implementation of DIA operations easy.

Thrill divides available system memory into three parts (by default equally): BlockPool memory, DIA operations memory, and free floating heap memory for user objects like *std::string*. All memory is tracked in Thrill by overloading *malloc()*, hence the user application needs no special allocators. Memory limits for DIA operations' internal data structures are negotiated and defined when executed during evaluation. The StageBuilder determines which DIA operations participate in a stage and divides the allotted memory fairly between them. It is important for external memory support that the operations adhere to these internal memory

limits, e.g. by correctly sizing their hash tables and sort buffers.

C. Details on the Reduce, Group, and Sort Implementations

Besides pipelining DIA operations, careful implementations of the core algorithms in the operations themselves are important for performance. Most operations are currently implemented rather straight-forwardly, and future work may focus on more sophisticated versions of specific DIA operations. Due to the generic DIA operation interface, these future implementations can then be easily plugged into existing applications.

1) Reduce Operations: ReduceByKey and *ReduceToIndex* are implemented using multiple levels of hash tables, because items can be immediately reduced due to associative or even commutative reduction operations $r : A \times A \to A$.

Thrill distinguishes two reduction phases: the pre-phase prior to transmission and the post-phase receiving items from other workers. Items which are pushed into the Reduce-DOp are first processed by the key extractor $k : A \to K$ or index function $i : A \to [0, n)$ (see Table I). The key space K or index space $[0, n)$ is divided equally onto the range of workers $[0, p)$. During the pre-phase, each worker hashes and inserts items into one of p separate hash tables, each destined for one worker. If a hash table exceeds its fill-factor, its content is transmitted. If two items with matching keys are found, they are combined locally using r.

Items that are received from other workers in the post-phase are inserted into a second level of hash tables. Again, matching items are immediately reduced using r. To enable truly massive data processing, Thrill may spill items into external memory during the post-phase. The second level of hash tables are again partitioned into k separate tables. If any of the k tables exceeds its fill-factor, its content is spilled into a File. When all items have been received by the post-phase, the spilled Files are recursively reduced by choosing a new hash function and reusing the hash table.

The pre- and post-phases use custom linear probing hash tables with built-in reduction on collisions. One large memory segment is used for p separate hash tables. Initially, only a small area of each partition is filled and used to save allocation time. When a hash table is flushed or spilled, its allocated size is doubled until the memory limit prescribed by the StageBuilder is reached.

2) Group Operations: GroupByKey and *GroupToIndex* are based on sorting and multiway merging of sorted runs. Items pushed into the Group-DOp are first processed by the key extractor $k : A \to K$ or index function $i : A \to [0, n)$, the result space K or $[0, n)$ is distributed evenly onto all p workers. After determining the destination worker, items are immediately transmitted to the appropriate worker via a Stream. Each worker stores all received items in an in-memory vector. Once the vector is full or heap memory is exhausted, the vector is sorted by key, and serialized into a File which may be swapped to external memory. Once all items have been received, the sorted runs are merged using an efficient multiway merger. The stream of sorted items is separated into subsequences with equal keys, and these sequences are delivered to the group function $g : \text{iterable}(A) \to B$ as a multiway merge iterator.

3) Distributed Sorting: The operation *Sort* rearranges all DIA items into a global order as defined by a comparison function. In the Link step on each worker, all local incoming items are written to a File. Simultaneously, a random sample is drawn using reservoir sampling and sent to worker 0 once all items have been seen. In the Main part, Thrill uses Super Scalar Sample Sort [**?**] to redistribute items between workers: worker 0 receives all sample items, sorts them locally, chooses $p-1$ equidistant splitters, and broadcasts the splitters back to all workers. These build a balanced binary tree with p buckets to determine the target worker for each item in $\lceil \log p \rceil$ comparisons. As Super Scalar Sample Sort requires the number of buckets to be a power of two, the tree is filled with sentinels as necessary. Items are then read from the File, classified using the splitter tree, and transmitted via a Stream to the appropriate worker. When a worker reaches its memory limit while receiving items, the items are sorted and written to a File. If multiple sorted Files are created, these are merged during the Push part.

Datasets with many duplicated items can lead to load balance problems if sorting is implemented naively. To mitigate skew, Thrill uses the global array position of the item to break ties and determine its recipient. When an item is equal to a splitter, it will be sent to the lower rank worker if and only if its global array position is lower than the corresponding quantile of workers.

IV. Experimental Results

We compared Apache Spark 2.0.1, Apache Flink 1.0.3, and Thrill using six micro benchmarks on the Amazon Web Services (AWS) EC2 cloud. Our benchmark and input set is based on HiBench [**?**], which we extended[2] with implementations for Flink and Thrill.

We selected five micro benchmark kernels: *WordCount*, *PageRank*, *TeraSort*, *KMeans*, and *Sleep*. WordCount is run on two different inputs. To focus on the performance of the frameworks themselves, we attempted to implement the benchmarks equally well using each of the frameworks, and made sure that the same basic algorithms were used. Spark and Flink can be programmed in Java or Scala, and we include implementations of both whenever possible. The code for Spark and Flink benchmarks was taken from different sources, all implementations for Thrill were written by us and are included in the Thrill C++ source code as examples. While we tried to configure Spark and Flink best possible, the complexity and magnitude of configuration

[2] http://github.com/thrill/fst-bench

options these frameworks provide make it possible that we may have missed some tuning parameters. For the most part we kept the parameters from HiBench. The experiments are run with weak scaling of the input, which means that the input size increases with the number hosts h, where each AWS host has 32 cores.

A. The Micro Benchmarks

Implementations of WordCount were available in Java and Scala from the examples accompanying Spark and Flink. The *WordCount1000* benchmark processes $h \cdot 32$ GiB of text generated by a C++ version of Hadoop's RandomTextWriter. There are only 1 000 distinct words in this random text, which we do not consider a good benchmark for reduce, since only very little data needs to be communicated, but this input seems to be an accepted standard.

Additionally, we ran WordCount on text data extracts from the CommonCrawl[3] corpus (September 2016). In *WordCountCC*, $h \cdot 128$ gzip-compressed "WET" archives were processed. Each WET archive is about 155 MiB compressed and extracts to approximately 392 MiB of plain text. This sums up to about $h \cdot 19.3$ GiB gzip-compressed text and $h \cdot 49$ GiB uncompressed text. Decompression of the archives is performed on-the-fly by the frameworks. Contrary to the synthetic WordCount1000 benchmark, this real-world text contains a huge number of words with few occurrences.

For *PageRank* we used only implementations which perform ten iterations of the naive algorithm involving a join of the current ranks with all outgoing edges and a reduction to collect all contributions to the new ranks. We took the implementation from Spark's examples and modified it to use integers instead of strings as page keys. We adapted Flink's example to calculate PageRank without normalization and to perform a fixed number of iterations. Thrill emulates a join operation using *ReduceToIndex* and *Zip* with the page id as the index into the DIA. The input graph for the experiments contained $h \cdot 4$ M vertices with an average of 39.5 edges per vertex, totaling $\approx h \cdot 2.7$ GiB in size, and generated using the PagerankData generator in HiBench.

TeraSort requires sorting 100 byte records, and we used the standard sort method in each framework. HiBench provided a Java implementation for Spark, and we used an unofficial Scala implementation[4] [**?**] for Flink. Hadoop's *teragen* was used to generate $h \cdot 16$ GiB as input.

For *KMeans* we used the implementations from Spark and Flink's examples. Spark calls its machine learning package, while Flink's example is a whole algorithm. We made sure that both essentially perform ten iterations of Lloyd's algorithm using random initial centroids, and we implemented this algorithm in Thrill. We fixed the number of dimensions to three, because Flink's implementation required a fixed number of dimensions, and the number of clusters to ten. Following HiBench's settings, Apache Mahout's GenKMeansDataset was used to generate $h \cdot 16$ M sample points, and the binary Mahout format was converted to text for reading with Flink and Thrill ($\approx h \cdot 8.8$ GiB in size).

The *Sleep* benchmark is used to measure framework startup overhead time. It launches one map task per core which sleeps for 60 seconds.

B. The Platform

We performed our micro benchmarks on AWS using h r3.8xlarge EC2 instances. Each instance contains 32 vCPU cores of an Intel Xeon E5-2670 v2 with 2.5 GHz, 244 GiB RAM, and two local 320 GiB SSD disks. We measured 86 GiB/s single-core/L1-cache, 11.6 GiB/s single-core/RAM, and 74 GiB/s 32-core/RAM memory bandwidth using a memory benchmark tool[5]. The SSDs reached 460 MiB/s when reading 8 MiB blocks, and 397 MiB/s when writing.

The h instances were allocated in one AWS availability zone and were connected with a 10 gigabit network. Our network measurements showed $\approx 100\,\mu$s ping latency, and up to 1 GiB/s sustained point-to-point bandwidth. All frameworks used TCP sockets for transmitting data.

We experimented with AWS S3, EBS, and EFS as data storage for the benchmark inputs, but ultimately chose to run a separate CephFS cluster on the EC2 instances. Ceph provided reliable, repeatable performance and minimized external factors in our experiments. Each EC2 instance carried one Ceph ODS on a local SSD, and we configured the Ceph cluster to keep only one replication block to minimize bandwidth due to data transfer. We did not use HDFS since Thrill does not support it, and because a POSIX-based distributed file system (DFS) provided a standard view for all frameworks. The other SSD was used for temporary files created by the frameworks.

All Spark implementations use the RDD interface. Support for fault tolerance in Spark and Flink incurred no additional overhead, because no checkpoints were written. By default checkpointing is deactivated and must be explicitly configured. All run-time compression was deactivated, and Spark was configured to use Kyro serialization.

We used Ubuntu 16.04 LTS (Xenial Xerus) with Linux kernel 4.4.0-31, Ceph 10.2.2 (jewel), Oracle Java 1.8.0_101, Apache Spark 2.0.1, Apache Flink 1.0.3, and compiled Thrill using gcc 5.4.0 with cmake in Release mode.

C. The Results

Figure 4 shows the median result of three benchmark runs for $h = 1, 2, 4, 8, 16$ hosts. We plotted the throughput per host in MiB/s where the input size in raw bytes, which is proportional to the number of items. Figure 5

[3] http://commoncrawl.org

[4] https://github.com/eastcirclek/terasort

[5] http://panthema.net/2013/pmbw/

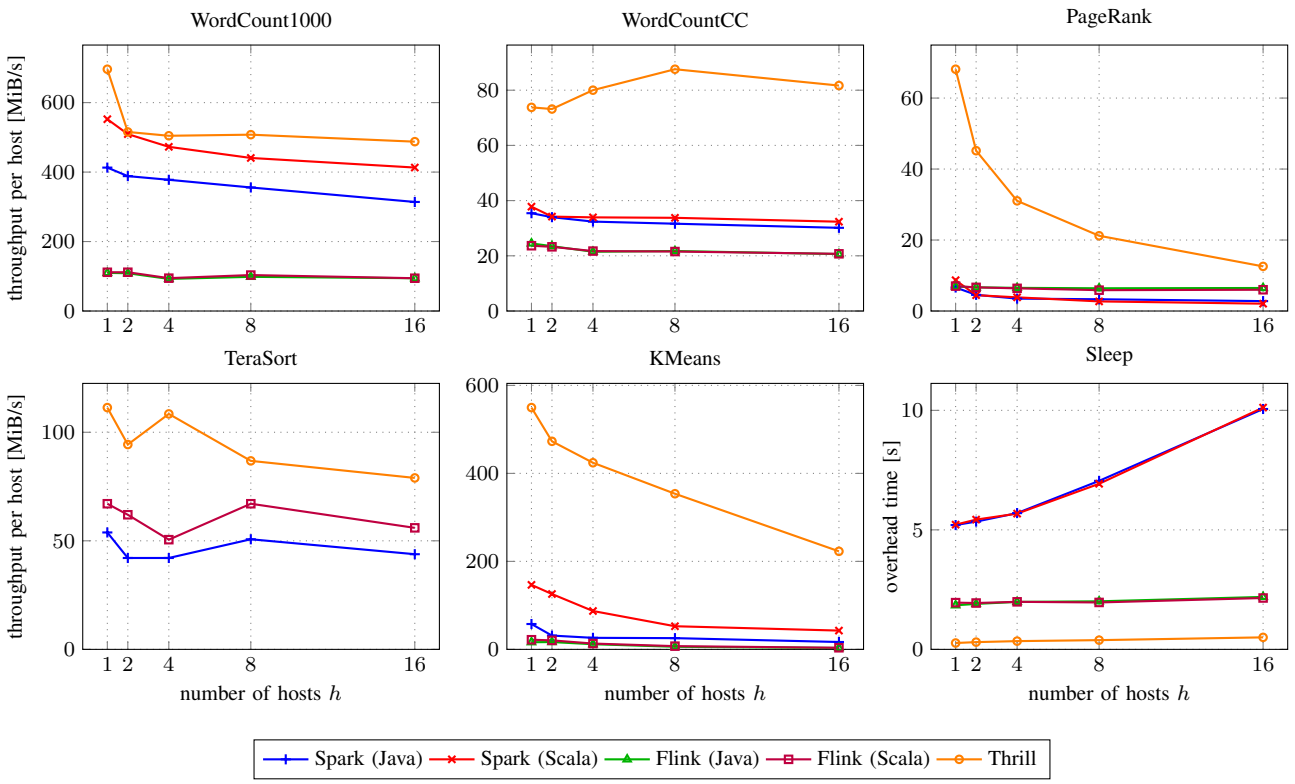

Figure 4. Experimental results of Apache Spark 2.0.1, Apache Flink 1.0.3, and Thrill on h AWS r3.8xlarge hosts

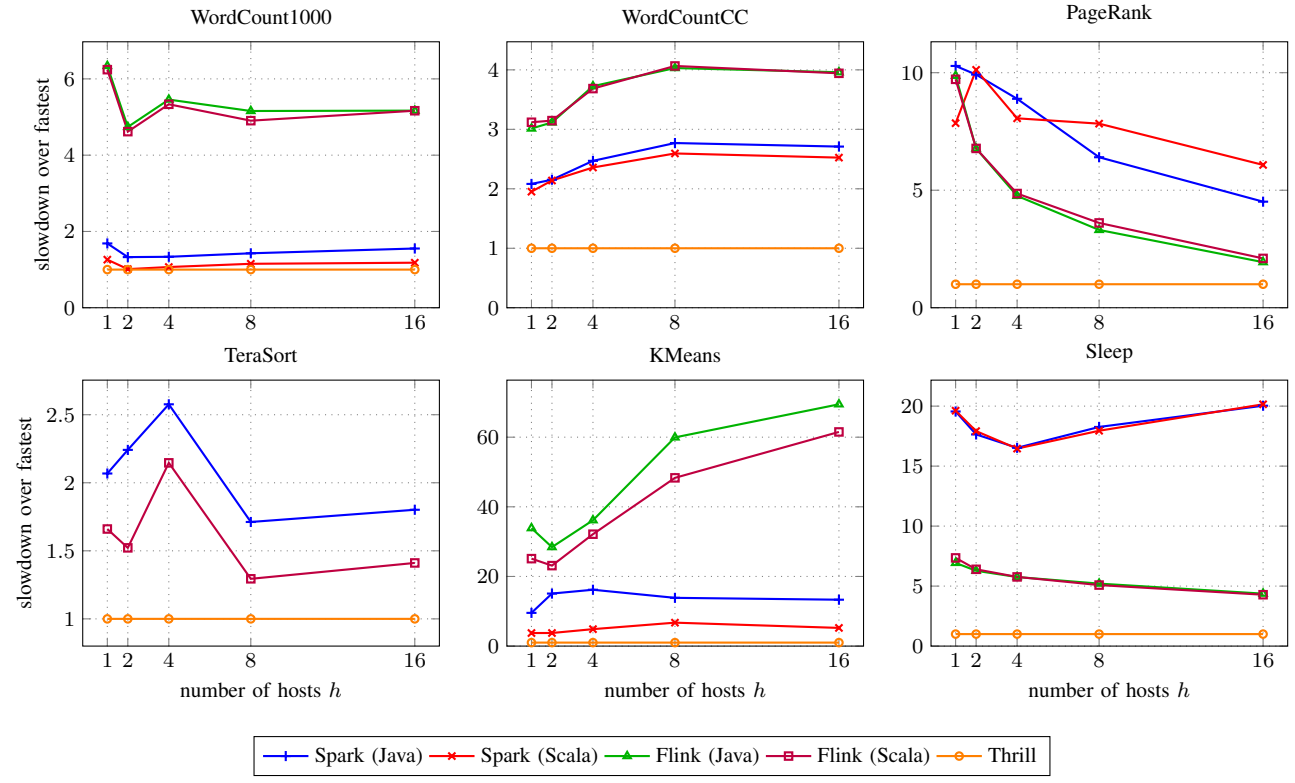

Figure 5. Slowdown of Apache Spark 2.0.1, Apache Flink 1.0.3, and Thrill on h AWS r3.8xlarge hosts over the fastest framework

shows the same results as Figure 4, except plotted as the slowdown in running time of each framework over the fastest. Additionally, we measured a performance profile of the CPU, network, and disk I/O utilization during the benchmarks using information from the Linux kernel, and show summarized results for $h = 16$ in Table II.

Thrill consistently outperforms Spark and Flink in all benchmarks on all numbers of hosts, and is often several times faster than the other frameworks. The speedup of Thrill over Spark and Flink is often highest on a single host, and grows smaller as network and disk I/O become bottlenecks.

In WordCount1000, the text is read from the DFS, split into words, and the word pairs are reduced locally. As only 1 000 unique words occur, the overall result is small and communication thereof is negligible. Thrill maximizes network utilization with 1 016 MiB/s via the DFS and uses 66% of the available CPU time for splitting and reducing. Spark also nearly maximizes the network with 931 MiB/s, and utilizes the CPU 65% of the running time. Flink is a factor 5.2 slower than Thrill in WordCount with 16 hosts, uses the CPU 73% of the time, and is not network bound. Thrill's reduction via hash tables are very fast, Spark is about the same, while Flink requires considerably more CPU time for the same task. With 16 hosts Thrill is network bound due to the DFS, and Spark (Scala) is only a factor 1.2 slower.

In WordCountCC, the DFS is no longer the bottleneck since the text archives are gzip-compressed. Surprisingly, the on-the-fly decompression was less overhead than expected, hence, this benchmark focuses on the reduction itself. As the reduced entries contain a dynamically allocated string, we believe that heap memory allocation strategies play the most important role in this benchmark. With 16 hosts Thrill is a factor 2.5 faster than Spark, and a factor of almost 4 faster than Flink.

In PageRank, the current rank values are joined with the adjacency lists of the graph and transmitted via the network to sum all rank contributions for the next iteration in a reduction. Hence, the PageRank benchmark switches back and forth ten times between high CPU load while joining, and high network load while reducing. Spark (Java) is a factor 4.5 slower than Thrill on 16 hosts, while Flink (Java) is a factor 1.9 slower. Flink's pipelined execution engine works well in this benchmark, and reaches 62% CPU and 15% network utilization. From the execution profile of Spark one can see that it does not balance work well between the hosts due to stragglers. Hence, each iteration takes longer than necessary. We believe Thrill's performance could be increased even further by implementing a *Join* algorithm.

In TeraSort, Spark is only a factor 1.8 slower and Flink a factor 1.4 than Thrill on 16 hosts. Spark reaches only 20% CPU and 42% network utilization on average, Flink 22% and 40%, and Thrill 21% and 43%, respectively. Flink's pipelined execution outperforms Spark in TeraSort, as was previously shown by an other author [**?**]. The implementations appear well tuned, however, due to the low CPU and network utilization, we believe all can be improved.

In the KMeans algorithm, the set of centroids are broadcast. Then all points are reclassified to the closest centroid, after which new centroids are determined from all points via a reduction. Like PageRank, the KMeans algorithm interleaves high local work and high network load (a reduction and a broadcast). Spark (Scala) is a factor 5.2 slower than Thrill on 16 hosts, Spark (Java) a factor 13, and Flink more than 60. We believe this is due to the JVM object overhead for vectors, and to inefficiencies in the way Spark and Flink broadcast the centroids. Flink's query optimizer does not seem to work well for the KMeans example accompanying their source package. Thrill utilizes the CPU 50% and the network 25% of the running time, while Spark reach 22% CPU and only 6% network utilization.

The Sleep benchmark highlights the startup time of the frameworks. We plotted the running time excluding the slept time in Figure 4. Spark requires remarkably close to $5 + h \cdot 0.4$ seconds to start up. Apparently, hosts are not started in parallel. Flink's start up time was much lower, and Thrill's less than one second.

V. Conclusion and Future Work

With Thrill we have demonstrated that a C++ library can be used as a distributed data processing framework reaching a similarly high level of abstraction as the currently most popular systems based on Java and Scala while gaining considerable performance advantages. In the future, we want to use Thrill on the one hand for implementing scalable parallel algorithms (e.g. for construction of succinct text indices) that are both advanced and high level. Thrill has already been

Table II
Input Size and Resource Utilization of Frameworks during Benchmarks

	Input Size	Spark (Scala)			Flink (Scala)			Thrill		
		CPU		Net	CPU		Net	CPU		Net
WordCount1000	$h \cdot 32$ GiB	50 s	65 %	931 MiB/s	251 s	73 %	195 MiB/s	43 s	66 %	1 016 MiB/s
WordCountCC	$h \cdot 19.3/49$ GiB	367 s	60 %	107 MiB/s	776 s	81 %	70 MiB/s	146 s	61 %	284 MiB/s
PageRank	$h \cdot 2.7$ GiB	383 s	28 %	36 MiB/s	284 s	62 %	154 MiB/s	37 s	17 %	260 MiB/s
TeraSort	$h \cdot 16$ GiB	73 s	20 %	425 MiB/s	65 s	22 %	396 MiB/s	42 s	21 %	425 MiB/s
KMean	$h \cdot 8.8$ GiB	81 s	22 %	64 MiB/s	190 s	4.3 %	27 MiB/s	35 s	50 %	250 MiB/s

The table shows the CPU utilization as seconds and percentage of total running time, and the average network bandwidth in MiB/s, both averaged over all hosts during the benchmark run with 16 hosts. TeraSort shows Spark (Java), as we have no Scala implementation.

used for more than five suffix sorting algorithms, logistic regression, and graph generators. On the other hand, at a much lower level, we want to use Thrill as a platform for developing algorithmic primitives for big data tools that enable massively scalable load balancing, communication efficiency, and fault tolerance.

While Thrill is so far a prototype and research platform, the results of this paper are sufficiently encouraging to see a possible development into a main stream big data processing tool. Of course, a lot of work remains in that direction such as implementing interfaces for other popular tools like Hadoop and the AWS stack, and creating frontends in scripting languages like Python for faster algorithm prototyping. To achieve practical scalability and robustness for large clusters, we also need significant improvements in issues like load balancing, fault tolerance and native support for high performance networks like InfiniBand or Omni-Path.

Furthermore, we view it as useful to introduce additional operations and data types like graphs and multidimensional arrays in Thrill (see also Section II-D). But, we are not sure whether automatic query plan optimization as in Flink should be a focus of Thrill, because that makes it more difficult to implement complex algorithms with a sufficient amount of control over the computation. Rather it may be better to use Thrill as an intermediate language for a yet higher level tool that would no longer be a plain library but a true compiler with a query optimizer.

ACKNOWLEDGMENT

We would like thank the AWS Cloud Credits for Research program for making the experiments in Section IV possible. Our research was supported by the Gottfried Wilhelm Leibniz Prize 2012, and the Large-Scale Data Management and Analysis (LSDMA) project in the Helmholtz Association.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The Stratosphere platform for Big Data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[4] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[5] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 293–307.

[6] P. Sanders, S. Schlag, and I. Müller, "Communication efficient algorithms for fundamental Big Data problems," in *IEEE International Conference on Big Data*, 2013, pp. 15–23.

[7] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[9] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 15–28.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in Spark," in *ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[12] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling Spark in the real world: performance and usability," vol. 8, no. 12. VLDB Endowment, 2015, pp. 1840–1843.

[13] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *preprint arXiv:1506.08603*, Jun. 2015.

[14] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[15] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay, "A comparison of Big Data frameworks on a layered dataflow model," in *9th International Symposium on High-Level Parallel Programming and Applications (HLPP'16)*, 2016, preprint arXiv:1606.05293.

[16] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251–267, 1994.

[17] P. Sanders and S. Winkel, "Super scalar sample sort," in *European Symposium on Algorithms, ESA'04*, ser. LNCS, vol. 3221. Springer, 2004, pp. 784–796.

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *New Frontiers in Information and Software as Services*, ser. LNBIP. Springer, 2011, vol. 74, pp. 209–228.

[19] K. Dong-Won, "Terasort for Spark and Flink with range partitioner," 2015. [Online]. Available: http://eastcirclek.blogspot.de/2015/06/terasort-for-spark-and-flink-with-range.html