A Distributed Stream Library for Java 8

Yu Chan

Doctor of Philosophy

University of York
Computer Science

August 2016

Abstract

An increasingly popular application of parallel computing is Big Data, which concerns the storage and analysis of very large datasets. Many of the prominent Big Data frameworks are written in Java or JVM-based languages. However, as a base language for Big Data systems, Java still lacks a number of important capabilities such as processing very large datasets and distributing the computation over multiple machines. The introduction of Streams in Java 8 has provided a useful programming model for data-parallel computing, but it is limited to a single JVM and still does not address Big Data issues.

This thesis contends that though the Java 8 Stream framework is inadequate to support the development of Big Data applications, it is possible to extend the framework to achieve performance comparable to or exceeding those of popular Big Data frameworks. It first reviews a number of Big Data programming models and gives an overview of the Java 8 Stream API. It then proposes a set of extensions to allow Java 8 Streams to be used in Big Data systems. It also shows how the extended API can be used to implement a range of standard Big Data paradigms. Finally, it compares the performance of such programs with that of Hadoop and Spark. Despite being a proof-of-concept implementation, experimental results indicate that it is a lightweight and efficient framework, comparable in performance to Hadoop and Spark.

Contents

List of Tables

List of Figures

Acknowledgements

Firstly, I would like to thank my supervisors Andy Wellings and Ian Gray for their support and attention to detail throughout my time at the University of York, as well as Neil Audsley for my experience in being part of the JUNIPER project. I would also like to thank everyone in the Real-Time Systems Group for all the discussions we had and the encouragement given. Finally, I would like to thank my family for being just an instant message away despite the thousands of miles separating us.

This work was supported by the JUNIPER project, which has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration (Grant No. 318763).

This work made use of the facilities of N8 HPC Centre of Excellence, provided and funded by the N8 consortium and EPSRC (Grant No. EP/K000225/1). The Centre is co-ordinated by the Universities of Leeds and Manchester.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2012 – 2016. This work has not been previously presented for an award at this or any other university. Except where stated, all of the work contained within this thesis represents my original contribution.

Parts of this thesis have previously been submitted or published in the following papers:

- Y. Chan, I. Gray and A. Wellings, *Exploiting Multicore Architectures in Big Data Applications: The JUNIPER Approach.* Proceedings of 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG 2014). Jan 2014.
- Y. Chan, A. J. Wellings, I. Gray and N. C. Audsley, *On the Locality of Java 8 Streams in Real-Time Big Data Applications.* Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2014). Oct 2014.
- Y. Chan, A. J. Wellings, I. Gray and N. C. Audsley, *A Distributed Stream Library for Java 8.* Submitted to IEEE Transactions on Big Data.

Chapter 1

Introduction

A trend in parallel computing is towards the storage and analysis of very large datasets, commonly referred to as *Big Data*. Madden [53] defines Big Data as data that is:

> *too big, too fast, or too hard for existing tools to process.*

In other words, these datasets are complex and difficult to process using traditional data processing techniques [54] such as databases, either due to the sheer scale of the data being produced, or tight timing requirements placed on the processing of the data.

In 2001, Laney [47] characterised Big Data as having three important dimensions, termed the *3Vs*:

- **Volume** – Indicates the size and scale of data, which is increasing exponentially. This can be attributed to the ease of creating data in the digital age. The following examples give an idea of how large the datasets are:
  - The Large Hadron Collider can output a raw data stream of approximately 1PB/s [84] which must be filtered before storage.
  - Aircraft, such as the Boeing 787, can output half a TB of data for each flight [35].

  It is estimated that by the year 2020 some 40ZB (40 trillion GB) of data will be produced in total, which is 300 times what was produced by the year 2005 [41].
- **Variety** – Conventional databases primarily deal with structured data, but Big Data is mainly unstructured and comes from a wide variety of sources and in many formats. Examples of data formats are Youtube videos (binary data), tweets (short strings of text) and Wikipedia encyclopedia entries (long text articles and binary data for images and audio).
- **Velocity** – A significant amount of data is produced on-the-fly (such as the Large Hadron Collider, aircraft monitoring equipment and stock market trading sessions [41]). It is sometimes necessary to analyse this data before storage, possibly to filter or compress it to reduce the size of data that needs to be stored.

More recently, this model has been extended to include other Vs:

- **Veracity** – Refers to the accuracy of Big Data, as inaccurate data is of no benefit [41]. This is better known by the common idiom "*garbage in, garbage out*".

- **Value** – Refers to the potential business opportunities and cost savings that arise from analysing Big Data [55].

- **Variability** – Refers to data (words, for example) whose meaning depends on the context in which it is used [58]. In some cases, context needs to be taken into account for accurate analysis of the data.

- **Visualisation** – Refers to the way Big Data is presented after analysis [58]. Big Data tends to be multidimensional and simple charts are usually inadequate for Big Data visualisation.

In some domains, requests for analytics and mining of stored datasets must be serviced sufficiently fast for the end user. In these situations, faster processing allows for potentially greater accuracy (i.e. by covering a larger historical dataset).

By definition, it is not feasible to use a single computer large enough to store and process Big Data, so computation is normally done on a cluster of commodity hardware or on a supercomputer. For a cluster of commodity hardware, it is most cost-effective to store data on hard disks. This is also done to increase the I/O bandwidth of the system, as disk speeds are even slower than main memory.

Using a cluster of multicore machines also adds another layer of parallelism (and thus complexity) for programmers to deal with, and the way computation and data is mapped to clusters has a large impact on the efficiency of the computation. To illustrate the complexity of mapping Big Data computations to a cluster, consider the example of a *word-count* [12] (the Big Data equivalent of *Hello World*) which outputs the number of words in a given text input. In a seven-node cluster, the input data for *word-count* can be stored in three nodes, the counting done on another three nodes, and the summing of partial counts done on the last node (see figure 1.1). However, this gives rise to several inefficiencies:

- Disk I/O bandwidth is low, as only three out of seven nodes are reading input data.

- The data and computation are not located in the same node, requiring the transfer of all data across nodes and resulting in high network usage.

- There is under-utilisation of the cluster: the summing node will be idle most of the time as it waits for the partial counts to arrive.

From the example above, it can be noted that data locality is a significant factor in the efficiency of Big Data computations. To make the computation more efficient, the mapping can be rearranged such that input data is partitioned across all nodes, with each node processing its local data and forwarding its partial count to a designated node (see figure 1.2). Efficiency is now improved:

- Disk I/O bandwidth is fully utilised, as all seven nodes are reading input data.

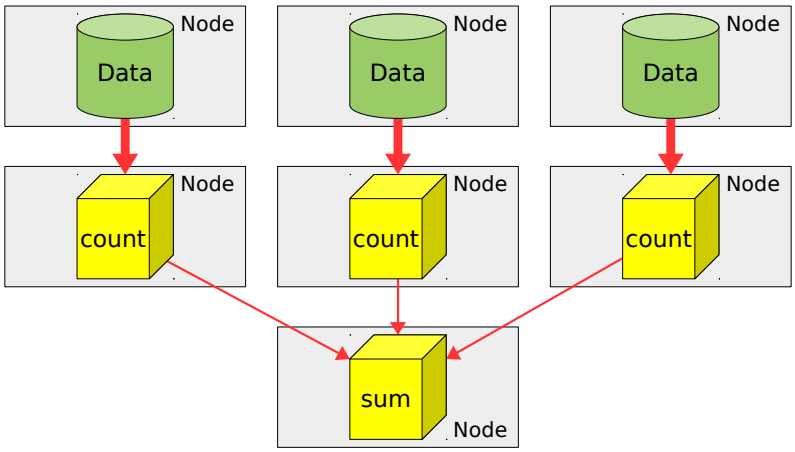


Figure 1.1: Naive mapping of *word-count* onto a 7-node cluster.

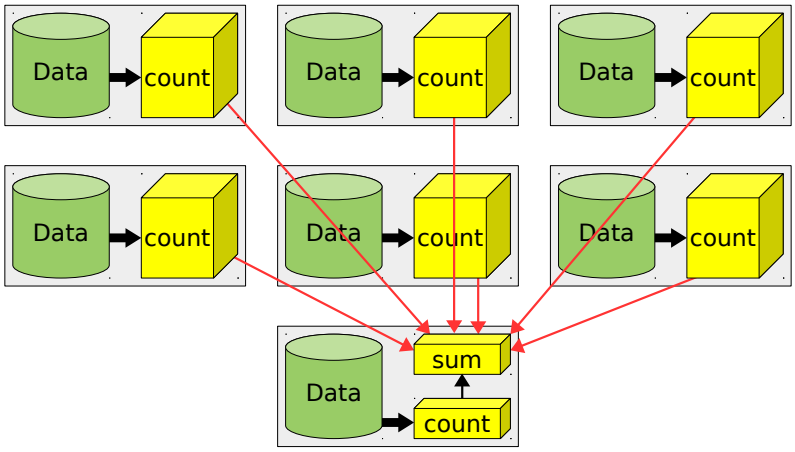


Figure 1.2: Improved mapping of *word-count* onto the same cluster.

- Since input data and counting tasks are on the same node, only the partial counts need to be sent over the network, thus using much less network bandwidth.

However, this introduces the issue of evenly distributing the input data and whether achieving this is feasible. This is something programmers and system designers did not have to be concerned about if the data size was small.

In summary, the main issues with programming Big Data systems are:

- How data is distributed. This includes partitioning, storage and retrieval.
- How computation is distributed.
- How data flows across the cluster.

This thesis is primarily concerned with the problem of programming Big Data systems and will thus be addressing these issues. It will focus on clusters with one disk per node.

1.1 Java

Recently, the 8th release of Java (henceforth called *Java 8*) added support for data-parallel computations to the language. This brings the base language closer to supporting Big Data processing.

Java is the base language for a number of popular Big Data frameworks, including Hadoop [4], Spark [8] and Storm [57]. Hadoop is a popular Big Data framework upon which many other higher-level frameworks such as Hive [6] and Mahout [7] are built.

The introduction of streams and lambda expressions in Java 8 has brought Big Data closer to the core language. Java 8 Streams allow programmers to think of data processing in terms of pipelines of operations, and complementing it are lambda expressions that simplify functional programming. However, this stream-based model is still not sufficient for Big Data for two main reasons. Firstly, streams are limited to computations within a single machine, and there is no mechanism for distributing computations over multiple machines. This is due to Streams being built on Java's Executor framework, which has no concept of distribution. Secondly, Java's default stream sources (such as collections and arrays) either store their data in-memory and thus cannot support very large datasets, or are not optimised for data-parallel computation.

This thesis addresses these issues and extends Java 8 with Distributed Streams (which introduce the concept of distributed computing on a cluster) and Stored and/or Distributed Collections (which introduce the concepts of accessing very large datasets on-demand and of datasets that are distributed over a cluster).

1.2 Thesis hypothesis

This thesis contends that the Java 8 Stream framework is inadequate to support all the requirements of programming Big Data systems. However, it is possible to extend the framework to meet all the requirements for programming Big Data systems, and still achieve performance comparable to or exceeding those of the popular Big Data frameworks (Hadoop and Spark, for example). Furthermore, this can be achieved in a way which retains compatibility with existing Java 8 software.

1.3 Thesis structure

The rest of the thesis is structured as follows:

- **Chapter 2** reviews the existing programming models and frameworks for Big Data.
- **Chapter 3** discusses the limitations of using the Java 8 programming model for Big Data processing and proposes a set of requirements for processing large datasets with Java 8.
- **Chapters 4 and 5** extend the Java 8 model to meet the requirements listed in chapter 3 and discusses the decisions and trade-offs made in developing the model.
- **Chapter 6** focuses on the implementation details of Distributed Streams.

- **Chapter 7** describes the tests done to compare the performance of Distributed Streams with those of Hadoop and Spark. It also evaluates the expressiveness of Distributed Streams.
- Finally, **chapter 8** suggests avenues of further research and improvements to the proposed model, and concludes the thesis.

Chapter 2

Programming Models for Big Data

This literature review first defines several terms relating to parallel computing that are used throughout the thesis, as well as a number of parallel computing concepts that are essential to understanding Big Data programming models. It then discusses existing approaches to solving the Big Data issues (section 2.5) mentioned in chapter 1. Finally, it explains the relevant new features provided by Java 8 (section 2.6).

2.1 Concurrency and parallelism

There is some confusion between the words "concurrency" and "parallelism" as they are often used interchangeably. For this research, these concepts will be defined as follows:

- *Concurrency* describes a program's structure. A concurrent program is split into parts that can potentially be executed simultaneously [72].
- *Parallelism* describes a hardware's architecture and the operating system (OS) that runs on it [62]. Parallel hardware usually refers to systems with multiple CPUs. Given a concurrent program, these CPUs can potentially execute the concurrent parts simultaneously. However, given a sequential program, one CPU will be executing it while the rest remain idle.

A parallel program is a concurrent program that was written with parallel hardware in mind. Depending on how it was written, it may assume a certain type of parallel hardware and fail to execute at all on others, or fall back gracefully and adapt itself to the given hardware.

A concurrent program, on the other hand, may not have been written with parallel hardware in mind, and may thus have latent faults which only manifest when executing on parallel hardware. These faults are generally difficult to trace and debug if the debugger is not aware of parallelism, as it may change the sequence of instructions that are executed.

Before the introduction of multicore hardware, concurrency was (and still is) achieved by multiplexing execution on a CPU. As a result, this type of concurrency is generally well-understood in the literature compared to concurrency on parallel hardware. Parallelism adds new challenges to efficiently programming such systems. For example, the choice of algorithms to use may be different, as those best suited for sequential execution are unlikely to parallelise well [72]. However, opinions diverge on when parallelism should

be considered in the software design process. Reinders suggests that it should be done early in the process rather than leave it as an afterthought [72], but Grama et al. suggest developing an optimised sequential implementation before parallelising it [37].

The use of functional programming concepts in parallel programming has gained popularity recently. This is due to functional programs being easier to parallelise. For example, new languages such as Scala [27] have both imperative and functional features.

2.2 Types of parallelism

To parallelise a program, one should be able to split it into parts that can execute in parallel, while keeping in mind that parallelism is limited by any sequential dependencies between the various parts of the program [51]. This section describes three ways a program can be split up, and gives examples of related languages in each case. (A single computer is assumed in these examples for simplicity.)

2.2.1 Task parallelism

Grama et al. define a task as a computation unit that can potentially be executed in parallel [37], and task parallelism as that obtained by tasks in a task-dependency graph [37]. Tasks that run in parallel may share input data but usually produce separate outputs [51].

There are several methods to achieve task parallelism, two of which are described in this section. If all CPUs execute the same piece of code, the program can be structured using conditional statements as follows, where each CPU executes its assigned task based on its identifier:

```
if (CPU == 0)
  taskA();
else if (CPU == 1)
  taskB();
```

The disadvantage of this method is that an assumption of hardware architecture is made – that there are at least two CPUs in this case – and the CPU mapping is hard-coded into the program.

An alternative method is to use threads, which are implemented by threading libraries on popular operating systems. Program execution begins with a single thread, and more threads are spawned as needed:

```
t1 = create_thread();
t2 = create_thread();
t1.run(taskA);
t2.run(taskB);
```

Examples of languages that provide support for threads are C (through the native C11 thread API and Pthreads), C++ (through the native `thread` API and Pthreads) and Java (through the `Thread` class). The threading library and OS map the threads to CPUs, thus programmers do not have to make assumptions about the number of CPUs in the architecture.

Though C, C++ and Java provide threads for task parallelism, programming with threads is considered low-level, and it is easy to introduce race conditions if not careful [50]. Although not all higher-level task-parallel languages eliminate race conditions, they help by allowing programmers to specify sections of code that can run concurrently and mapping this to threads automatically, thus freeing the programmer from having to deal with thread handling. An example of such a language is Cilk [71], which provides keywords to indicate that a function can be spawned concurrently (`cilk`), to spawn a function concurrently (`spawn`) and to wait for spawned functions to return (`sync`). If these keywords are removed, the result is still valid sequential C code. Cilk does not automatically prevent race conditions in concurrent functions, thus programmers still need to use locks to synchronise execution where necessary.

2.2.2 Data parallelism

Whereas task parallelism splits a program into multiple tasks, data parallelism splits the data that a program operates on and processes each part using identical tasks [51]. Data parallelism is a special case of task parallelism. Consider the following task that populates the specified range of an array:

```
void task(int array[], int from, int to)
{
  for (int i = from; i < to; i++)
    array[i] = ...;
}
```

Parallelising this task with the first method described in Section 2.2.1 gives:

```
int array[100];
if (CPU == 0)
  task(array, 0, 50);
else if (CPU == 1)
  task(array, 50, 100);
```

or simply:

```
int array[100];
task(array, CPU, CPU + 1);
```

if there are the same number of CPUs as array elements. If the workload and number of CPUs are not known in advance, the code would then be:

```
int array[N];
int from = CPU * N / NumCPUs;
int to = (CPU + 1) * N / NumCPUs;
task(array, from, to);
```

The granularity of the task also needs to be considered in data parallelism [51]. If each task is given a similar amount of data to operate on but runs with different execution times, the workload distribution is uneven and, towards the end, some CPUs will be idle while others are still working. This can be mitigated by reducing the granularity of the task, giving it smaller chunks of data to operate on (for simplicity, a granularity of 1 is used in the example), while load balancing the increased number of tasks over the CPUs:

```
int array[N];
int index = 0;
while (index < N && (index = getNextIndex()))
  task(array, index, index + 1);
```

This, however, introduces the need to get the next piece, or pieces, of work to operate on. There are several methods to implement this. The simplest is to increment a counter atomically or by using a critical section:

```
int count = 0;
int getNextIndex()
{
  enter_critical_section();
  int result = count++; // atomic operation
  exit_critical_section();
  return result;
}
```

Since this requires the use of atomic CPU instructions or critical sections, there is contention between CPUs as `getNextIndex` is called frequently. An alternative is to use work stealing [72] – each CPU has a queue of tasks to execute, and when a CPU runs out of tasks in its queue, it transfers a task on another queue to its own.

Besides Big Data, there are many other applications for data parallel computing, including simulating cellular automata, solving n-body problems and collision detection [59].

Since data parallelism is a special case of task parallelism, all languages that support task parallelism also support data parallelism. However, some languages simplify data parallel programming by supporting certain language constructs. For example, a `forall` construct that parallelises a similarly coded sequential `for`-loop can be found in High Performance Fortran (HPF) [74] (as `FORALL`) and Cilk Plus [23] (as `cilk_for`). HPF also provides the `PURE` keyword for the declaration of pure functions – functions with no side effects – and limits function calls inside `FORALL` constructs to those that call pure functions. This eliminates any interference when multiple such functions are executed in parallel.

2.2.3 Stream parallelism

Stream parallelism, or pipelined decomposition, is a combination of task and data parallelism [51]. The processing of each piece of data that enters the pipeline is broken down into several tasks which can execute in parallel, but with data being fed sequentially from one task to another in a certain order [37]. This type of parallelism is useful for event and video processing where low latency is needed [15].

An example of a stream parallel language is StreamIt, a Java-like language [82]. In StreamIt, the programmer defines stream objects, the simplest of which are filters (tasks), and a pair of stream objects can be connected via tapes (infinite sequences) which store output from one object and feed them as input into the other. The other type of stream object is the composite stream, which consists of multiple filters. StreamIt limits composite streams to three predefined types – pipelines (a number of stream objects connected linearly), splitjoins (input is split among a number of stream objects) and feedback loops

(output of a stream object is fed back into the input after being processed by another stream object).

2.3 Programming and memory models

The architecture of a system determines the memory model supported at the lowest level (as one can be expressed in terms of the other). The majority of parallel programming models are focussed on either shared memory or message passing communication mechanisms.

There are many ways to group parallel programming models, but a classification that has far-reaching implications is whether a model is *fragmented* or *global-view*. In a fragmented model, programmers are required to explicitly split algorithms into tasks and map them onto the system [21]. In a global-view model, programmers do not need to do this; if necessary, the mapping is specified separate from the algorithm [21]. Fragmented models tend to be low-level and difficult to write code for, but are expressive. On the other hand, global-view models tend to be high-level and easy to write code for, but are often less flexible and more difficult to implement (the implementation has to handle the splitting and mapping of tasks). In other words, programmers assume the burden of complexity in a fragmented model, while framework or library implementers have to handle it in a global-view model instead.

2.3.1 Shared memory mechanisms

Shared memory is a popular abstraction of symmetric multiprocessing (SMP) architectures, and many imperative languages such as C and Java use this abstraction implicitly. In shared memory, the programmer sees a system with a single address space that all CPUs can directly access [37]. This simplifies communication between CPUs, as it is not necessary to make copies of data to be transmitted, since other CPUs can access the original contents with just a memory address.

However, this gives rise to memory consistency issues when multiple CPUs attempt to modify memory at the same address. This is handled by using atomic CPU instructions such as *swap* (the `XCHG` instruction on x86) and *compare-and-swap* (`CMPXCHG` on x86). On multicore architectures that support out-of-order execution (e.g. x86), memory fences [1] are also used to maintain consistency. These impose a particular order on memory accesses by different CPUs.

Shared memory systems can be categorised by the latency of memory access. In a uniform memory access (UMA) system, all CPUs incur the same latency for accessing any main memory location. In a non-uniform memory access (NUMA) system, CPUs incur different latencies when accessing different memory locations. Current multicore architectures are usually cache coherent NUMA (ccNUMA) systems, where the hardware transparently handles cache coherency issues that arise from multiple CPUs accessing the same memory location at the same time. The advantage of shared memory's simplicity outweighs any variation in memory access latencies.

However, shared memory does not scale well because it becomes more difficult to

maintain cache and memory coherency as the number of CPUs in multicore architectures increases. On distributed memory architectures, it is restricted to local memory only, as remote memory locations are not directly accessible. There have been attempts to extend shared memory to distributed architectures (called distributed shared memory) by adding support in operating systems or libraries (such as the Grappa distributed shared memory runtime [60]), but this is complex and involves implementing memory coherency in software.

OpenMP

OpenMP is a shared memory, global-view model API for structured parallel programming. It is implemented as a runtime which manages a pool of threads, and a set of compiler directives which parallelises blocks of code over the thread pool [26]. Exact syntax differs between languages; in C, OpenMP supports the following features:

- Task parallelism via the `#pragma omp parallel` directive on a statement to indicate that it is a parallel region and should be executed in parallel by threads in the pool, and the `#pragma omp parallel sections` and `#pragma omp parallel section` directives to indicate statements that a thread can run in parallel. With the introduction of OpenMP 3.0, programmers can specify tasks using the `#pragma omp task` directive. A task is either executed by the current thread immediately or passed to another thread for execution later.

- Data parallelism via the `#pragma omp parallel for` directive on a for-loop.

- Thread synchronisation via the `#pragma omp critical` directive to specify a critical section and the `#pragma omp barrier` directive to specify a barrier in a parallel region.

The shared memory model that OpenMP relies on, however, prevents such programs from scaling efficiently to distributed systems. Thus, in a system with distributed memory, OpenMP may be used within regions of shared memory while another model handles remote memory accesses.

2.3.2 Message passing mechanisms

Shared memory does not scale to distributed memory architectures due to CPUs not having direct access to all memory locations. The message passing mechanism augments shared memory in these systems, with communication between CPUs being achieved by sending and receiving messages via an interconnect [37].

An extreme example of a language with a built-in message passing mechanism is Smalltalk, which interprets a binary operation expression (*`operand1`* `operator` *`operand2`*) as passing a message `operator` consisting of *`operand2`* to *`operand1`*. Message passing mechanisms can also be found the POSIX socket API (primarily used in POSIX-compliant C programs) and the Remote Method Invocation API (used in Java programs) to work around their shared memory limitations.

Due to the architecture of commodity hardware clusters (typically SMP machines connected by Ethernet), Big Data programming models are usually a hybrid of both shared memory and message passing mechanisms. Computation and communication within each machine is done with shared memory, while communication between machines is message-based using packets.

MPI

The Message Passing Interface (MPI) is the de facto standard for communication via message passing, especially in high-performance computing [26], but also in clusters of commodity hardware called Beowulf clusters [43]. It exposes a fragmented programming model and is commonly used with a number of co-operating processes which run in parallel and do not share any part of their address space. These may be run on distributed systems, including heterogeneous ones. Several MPI library implementations (OpenMPI and MPICH for example) exist for common programming languages such as C and Java.

MPI supports sending and receiving arrays of both primitive and derived data types. Primitive data types include byte arrays and integers, while derived data types include structs and vector types. Programmers are required to define all derived data types and commit them to MPI before using them. Some MPI libraries support sending and receiving language-specific data types. For example, Java objects can be transmitted with the MPJ Express library's [79] *object* data type, thus handling serialisation and deserialisation of objects automatically for the programmer.

MPI defines one-to-one and collective operations between processes, with collective operations expressible as a sequence of one-to-once operations. One-to-one operations include both blocking and non-blocking sends and receives. Collective operations between a group of processes include:

- Barriers, where all processes synchronise at a particular point in the program.
- Broadcasts, where one process sends the data to all other processes.
- Scatters (and the inverse – gathers), where one process sends different parts of an array to all other processes.
- All-to-all operations, where each process gathers all parts of an array from all other processes, resulting in each process having a copy of the whole array.

Version 2 of the MPI specification added support for one-sided communication [26], which is for remote memory access via low-latency network connections such as InfiniBand.

MPI requires that the programmer deal with low-level issues such as network topology, the number of communicating processes [36] and fault-tolerance. By default, all processes terminate on encountering an error [38], so programmers have to modify this setting and handle errors manually.

However, MPI is not immune to the consequences of shared memory. With the proliferation of multicore architectures, MPI implementations have to address the issue of thread safety in multithreaded programs [26]. During MPI initialisation, a program can specify one of the following four thread safety levels it requires:

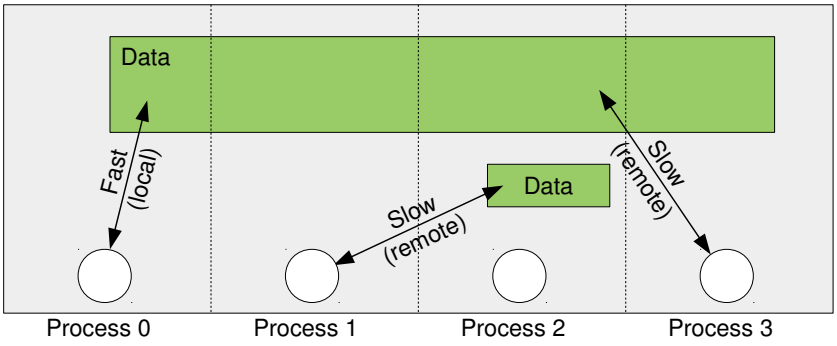


Figure 2.1: PGAS two-level memory model.

- `MPI_THREAD_SINGLE`, if the program is single-threaded.
- `MPI_THREAD_FUNNELED`, if subsequent MPI calls are made only from the thread performing the MPI initialisation.
- `MPI_THREAD_SERIALIZED`, if more than one thread may make MPI calls and their calls will be serialised by the program.
- `MPI_THREAD_MULTIPLE`, if more than one thread is involved and no serialisation is performed. That is, full thread safety is required from the implementation. For correct behaviour, the program still has to ensure that there are no race conditions resulting from MPI calls which refer to the same piece of data [39].

The MPI implementation, however, is free to support any of the thread safety levels. The program has to check the actual thread safety level that MPI has been initialised with, and handle the cases when it is not what the program expects. Gropp and Thakur conclude that making a MPI implementation thread safe is a difficult task and there is currently no way to test an implementation's claim that it is thread safe [39].

2.3.3 Partitioned global address space model

The partitioned global address space (PGAS) model is an attempt to address the scalability issues of shared memory and the different memory access semantics of message passing. It extends the shared memory mechanism to distributed memory architectures with two distinct levels of memory (fast local memory and slow remote memory; see figure 2.1). It defines a single address space, and each process or thread is assigned a local partition of it [77]. Data can either be allocated globally or locally, with global data distributed over the partitions and accessible from other processes [77]. This makes the PGAS model suitable for data parallel computations.

The PGAS model still suffers from a number of issues: the lack of support for task parallel computations and heterogeneous systems [77]. This prompted the development of the asynchronous partitioned global address space (APGAS) model, which adds the ability to spawn new tasks and define locations of coherent memory [77]. This allows task parallel computations to be performed and locations of tasks to be specified. Currently, most operating systems do not support PGAS or APGAS natively. Thus it has to be

implemented in a library (an example of such a library is GASNet [49]) or mapped from a new programming language.

Fatahalian et. al. propose an extension to the PGAS model called Sequoia, to support architectures with more than two levels of memory [34]. Sequoia allows a programmer to control the movement of data within the memory hierarchy of a system at a high level [34], and splits a machine's memory into a tree of mutually exclusive memory spaces, with memory spaces further from the root faster than those nearer the root. Each memory space has a corresponding processor, and it can only access memory from its memory space. To access memory outside its memory space, the required data has to be transferred between memory spaces first. Programs consist of a tree of isolated tasks, and the runtime maps each task to a memory space. Child tasks can potentially run on a child memory space. This model is suitable for parallel divide-and-conquer algorithms, with child tasks being smaller instances of parent tasks.

In the last decade several new languages which use the APGAS model have been developed to simplify parallel programming for distributed systems. The remainder of this section reviews two such languages, Chapel and X10.

Chapel

The Cascade High Productivity Language (Chapel) is a parallel language developed by Cray and designed to target a wide range of architectures from multicore and distributed computing to supercomputers. Its main goal is to simplify parallel programming while being as robust and efficient as widely used programming models [21]. It has a multi-layered design, where higher-level constructs are built upon lower-level ones.

A location with SMP-like processing and memory resources is known as a locale in Chapel [21], and tasks can be assigned to it for execution. A locale's address space is part of a global logical address space, accessible from other locales. To synchronously execute a task on a particular locale the `on` *`locale`* `do` statement can be used. Similarly, to ensure that execution takes place on a locale (or locales) where the required variable is found, the `on` *`var`* `do` statement can be used.

For parallelism, Chapel supports task parallelism, as well as data parallelism which is internally mapped onto task-parallel constructs. It supports the common sequential constructs, such as `if`, `for` and `while`. For task parallelism, the following primitive constructs are provided:

- A `begin` keyword that precedes a statement asynchronously spawns a task to execute the statement.
- A `sync` keyword that precedes a statement executes the statement and waits for it and all its spawned tasks to complete.

Higher-level task parallelism constructs include the following:

- A `cobegin` keyword that precedes a block statement asynchronously spawns a task for each child statement and waits for them to finish. Any tasks that are spawned by the child tasks are not waited on.

- A `coforall` statement spawns a specified number of tasks which execute the body asynchronously and waits for them to finish.

For data parallelism, the following constructs are provided:

- Domains and arrays, which are closely related. A domain defines a set of indexes, and an array is defined over a domain. Domains can be distributed over locales by specifying a mapping for each domain index; arrays defined over such domains will have their data distributed accordingly.

- A `forall` statement iterates over an array or domain, and distributes the computation over a number of tasks. Iterating over a distributed array causes remote tasks to be spawned automatically to handle the relevant parts of the array.

- Chapel defines a number of reduction operations that can be performed on an array with the *`op` `reduce` `expr`* expression. It is also possible to write custom reductions.

Synchronisation in Chapel can be achieved in two ways:

- With the `sync` keyword preceding a statement, mentioned above.

- With sync-types. If a variable is declared with the sync keyword, it behaves as a single-valued buffer. Reading an empty variable blocks until another task writes a value to it, and writing to a variable blocks if the last written value has not yet been read.

Compilation is done in stages – Chapel code is first translated into C, then compiled into machine code for the target architecture.

As Chapel is a relatively new language, some of its planned features, such as hierarchical locales, are still in development. Thus, a Chapel program may not yet run efficiently on ccNUMA architectures, since such architectures cannot be adequately represented with just single- or multi-dimensional arrays of locales.

X10

X10 is a distributed parallel language developed by IBM. It is also object-oriented, with Java having influence over its syntax [31].

X10 defines a *place*, similar to Chapel's locale, as a location with SMP-like characteristics, and an activity as a task [22]. Once spawned, an activity cannot be stopped. An activity is not allowed to access remote data; it has to spawn activities at the remote place to do that instead [22]. The `at(`*`place`*`)` *`A`* construct synchronously executes an activity A in a specified place. When invoked, all dependent objects and variables of A are copied to the remote place [76]. If A is a statement, any changes made to data in place are ignored and execution resumes at the original place upon completion. If A is an expression, any changes made to data in place are ignored, except for the the value of A which is returned, and execution resumes at the original place upon completion.

Task parallelism in X10 is achieved with the following:

- The `async` keyword that precedes a statement asynchronously spawns an activity which executes the statement. This corresponds to Chapel's `begin` keyword.
- The `finish` keyword that precedes a statement executes the statement and waits for all its spawned activities to complete. This corresponds to Chapel's `sync` keyword.

The following constructs are provided for data parallelism:

- *Regions* and *arrays* for distributed computing. Similarly to Chapel, a region describes how data in a distributed array is stored. Arrays are indexed by a tuple of integers, or *points*, and each has a corresponding region which holds the points in the array [22]. A distribution stores the place of each distributed array element [22].
- The `ateach` statement iterates over a distributed array and spawns an activity for each element of the array at the corresponding place.

Synchronisation in X10 can be achieved with the `atomic` keyword, which lets only one activity execute the statement that follows it.

Compilation of X10 code also takes place in stages. X10 code can be compiled to machine code via translation into C++, or to Java bytecode via translation into Java.

Currently, all places are assumed to be homogeneous. This may change in future versions of X10 with planned support for accelerators.

2.4 Parallel computation on clusters

Historically, supercomputers were used for computations requiring more processing power. With the increasing performance of commodity hardware, a cluster of commodity PCs has become more cost effective compared to proprietary and expensive supercomputers. Clusters are also useful for fault tolerance, as the entire computation does not have to be repeated if a computer (or node) fails, though this requires software support.

Clusters are more difficult to program efficiently, due to the two-level view of memory in the system. Shared memory does not extend beyond a single node, so message passing is needed for inter-node communication. Furthermore, communication between nodes is slower than within a node, hence existing algorithms that assume a single level of memory (usually shared memory) need to be modified to favour local over remote communication.

2.4.1 Parallel computation paradigms

A number of paradigms, or styles, can be found in cluster-based parallel computing [19]. This section discusses the popular ones related to Big Data computing.

Task farming

Also known as master-slave, this paradigm designates a master node which is responsible for splitting the computation into smaller ones for the other (slave) nodes to perform [19]. Upon completion of the smaller computations, the partial results are sent to the master node, which combines them into the final result (see figure 2.2). Communication is thus

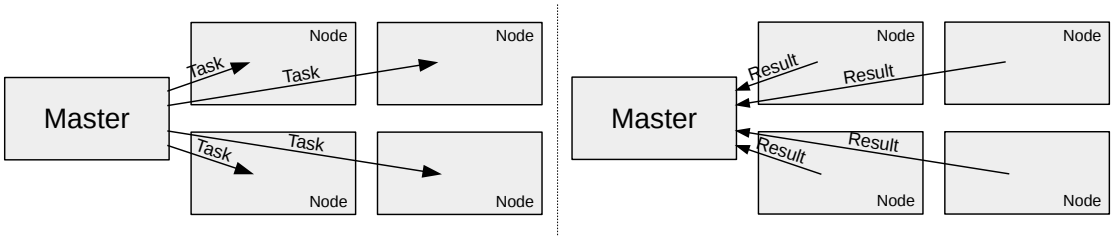


Figure 2.2: Task farming paradigm. Left: master splits computation into tasks and hands them to other nodes. Right: master receives partial results from nodes.

between the master and slave nodes only. This makes the master node a single point of failure, though there are variations of the paradigm that address this by defining a set of master nodes instead of a single node [19].

The task farming paradigm is useful for data parallel computations without data dependencies, especially on large datasets distributed across the cluster. This is achieved by having slave nodes perform the same task but on different data. For example, a distributed *word-count* fits well into the task farming paradigm.

Single program multiple data

In the single program multiple data (SPMD) paradigm, there is no centre of control, and all nodes run the same program on different data [19]. Data parallelism is better supported in SPMD as communication between all nodes is possible, allowing for more complex operations such as barrier synchronisation and broadcasting (see figure 2.3). However, this makes the system less fault tolerant, as a node failure can potentially cause all other nodes to stall [19]. A well-known example of a programming model that uses the SPMD paradigm is MPI.

Actor

The actor paradigm involves entities called actors that communicate with each other via asynchronous message passing and perform actions in response to the messages they receive [50]. Actors can either be supported implicitly in a language (such as Erlang [32]) or implemented in a library (such as Akka [83]). Message passing libraries such as MPI and ZeroMQ [42] can also be used to implement the actor paradigm.

Data pipelining

The data pipelining paradigm is based on stream parallelism, and is useful if the computation can be broken down into multiple tasks (or stages) which can execute concurrently [19]. A data item flows through each stage sequentially, but at any time each stage is processing a different data item. Data pipelining can be found in Unix shell commands that use pipes; the output of one process is fed into the input of another.

This paradigm can be mapped to a cluster by having nodes or groups of nodes execute different stages, or by having processes or threads within a node execute different stages and replicating this configuration on other nodes. The former mapping is less complex

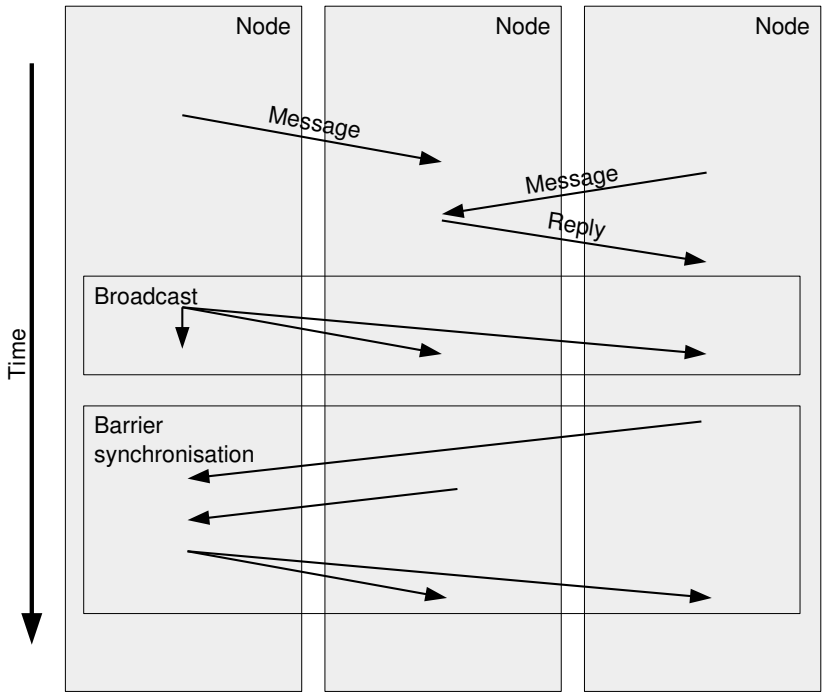


Figure 2.3: SPMD paradigm. Shows different types of communication between nodes.

than the latter, but suffers from having to move data across nodes, which is slower than local memory access.

2.4.2 Load balancing

An important issue in cluster computing is balancing the workload of each node [37]. Ideally, each node should be given an equal workload, but it is often not feasible to determine the amount of input data that corresponds to these workloads. Thus, methods such as work stealing are employed, where idle nodes are allowed to claim part of another node's workload.

Load balancing is also desirable in Big Data computations, but it is complicated by the inefficiency of moving large parts of the dataset to achieve this. This has led to modified work stealing algorithms that take data locality into account [86]. Load balancing is also related to two of the three Big Data issues mentioned in chapter 1 – the distribution of data and distribution of computation. Both these issues affect the workload of nodes relative to one another.

2.5 Big Data programming models and frameworks

This section reviews several Big Data programming models and frameworks developed to address the issues mentioned in chapter 1. It can be observed that there are similarities between most of the programming models discussed in this section:

- Big Data programming models are global-view rather than fragmented. They support data parallelism well, but have little or no support for task parallelism. This

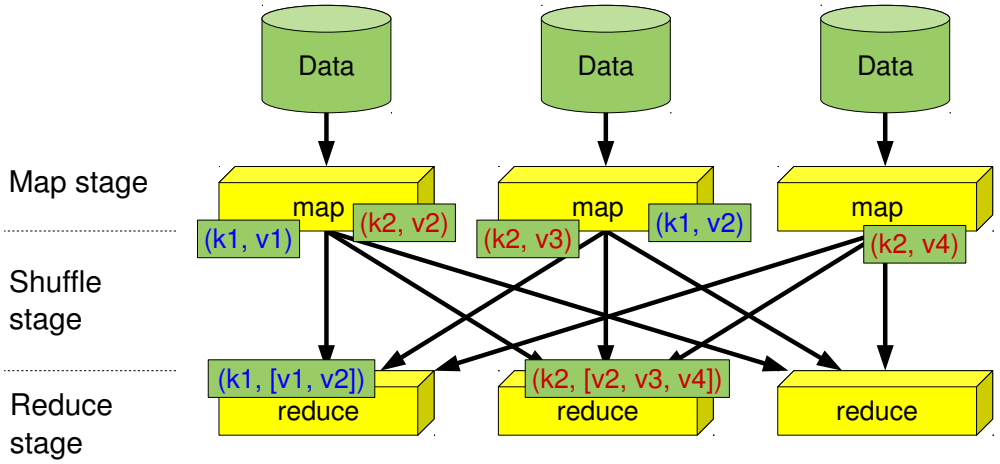


Figure 2.4: MapReduce computation stages, showing the movement of key-value pairs. Arrows indicate data flow.

makes them less expressive but easier to program for its intended uses (Big Data). The restrictions give implementations room to address the Big Data-specific issues.

- Big Data programming models are designed for processing large volumes of data and thus emphasise data locality.
- Since the chance of a hardware failure increases with larger clusters, Big Data programming models that are used in production environments are designed to be fault tolerant with minimal additional effort from the programmer.

2.5.1 MapReduce and Hadoop

MapReduce [25] is a Big Data programming model developed by Google. Its popularity is due to Hadoop [4], an open source MapReduce framework for Java.

A MapReduce computation consists of three stages, illustrated in figure 2.4:

1. *Map* stage: Input data is processed by a set of mappers, which output key-value pairs as a result.
2. *Shuffle* stage: The key-value pairs are collected over a set of reducers, with the same keys sent to the same reducer. For each key, the values are collected and sorted in a key-*value-list* pair.
3. *Reduce* stage: Each reducer processes all given key-*value-list* pairs and optionally outputs a result.

Mappers and reducers are implemented by the programmer, while the *shuffle* stage procedure is fixed and can be optimised by a MapReduce framework.

This model of computation is more efficient if each mapper has local access to its partition of the input data. Hadoop facilitates this by introducing the Hadoop Distributed File System (HDFS) [11], a locality-aware distributed file system. Before a MapReduce computation begins, input data is copied to HDFS. HDFS splits a file into blocks (usually 64MB or 128MB) which are distributed across the cluster. For fault tolerance, each block

is replicated 3 times by default and each copy stored on a different node. During the *map* stage in Hadoop, a mapper reads from local file blocks as far as possible, and the output from each reducer is written back as a separate file. HDFS does not allow the contents of a file to be overwritten, thus it is not POSIX-compliant.

Hadoop uses a task farming paradigm for computation, and designates several roles to the nodes in a cluster. A Hadoop cluster has one *JobTracker* and *NameNode* each. The NameNode stores information on the distributed filesystem, such as on which machine pieces of data are located. The JobTracker receives MapReduce jobs from client applications, consults the NameNode for the location of data, and distributes work to the *TaskTracker*s in a way that data movement over the network is minimised. The TaskTrackers, each running on a machine in the cluster, receive work in the form of map and reduce tasks, and spawns a separate JVM for each task so that an error in a task does not affect other tasks. A *DataNode* stores HDFS blocks as indexed by the NameNode. If a map task requires data not found in the DataNode on the local machine, it retrieves the data from an appropriate DataNode.

Listing 2.1 illustrates a *word-count* Hadoop program [12]. In the `main` method, a Hadoop job is configured to use a mapper (`TokenizerMapper`) that emits *(word, 1)* key-value pairs from each line, and a reducer (`IntSumReducer`) that adds up the values from the same key (word).

Though popular, there are major disadvantages of using Hadoop:

- The framework has a very high latency as it is optimised for batch processing [61].
- The user does not have fine-grained control over how data is distributed in HDFS. Thus, optimisations such as keeping data in a small subset of nodes to reduce data transfer in the *shuffle* stage are not possible. (Section 2.5.4 describes extensions to Hadoop that allow more control over data partitioning.)
- Input data that is not already in HDFS incurs an extra step to copy them into the filesystem. Furthermore, files in HDFS are write-once, so changing the contents of a file is inefficient.
- Due to Hadoop being a general purpose Big Data framework, it has to support a wide range of deployments, resulting in many configuration variables. This makes Hadoop complex to configure.
- Hadoop was designed to be portable, but this comes with inefficiencies such as not using the OS's filesystem cache [78]. It also makes assumptions about system behaviour such as optimised scheduling for concurrent disk accesses, leading to performance that is heavily system-dependent [78].

2.5.2 Stream-based models

Though suitable for many applications, the MapReduce model is inflexible due to the fixed stage sequence. More recently, focus has shifted to in-memory stream-based models for Big Data processing, with the Spark and Storm frameworks being prominent examples of such models.

```
public class WordCount {
  public static class Map extends MapReduceBase
      implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter) {
      StringTokenizer tokenizer = new StringTokenizer(value.toString());
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase
      implements Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter) {
      int sum = 0;
      while (values.hasNext())
        sum += val.next().get();
      output.collect(key, new IntWritable(sum));
    }
  }

  public static void main(String[] args) {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
  }
}
```

Listing 2.1: Hadoop (version 1.2.1) *word-count* example.

Spark

Spark improves on MapReduce by exposing a pipeline-based programming model for data processing. Instead of implementing mappers and reducers, programmers specify a pipeline of smaller and simpler operations on a *Resilient Distributed Dataset* (RDD). An operation either returns a new RDD (a *transformation*, which can be chained) or returns a value (an *action*, which terminates the pipeline).

Listing 2.2 illustrates the *word-count* example for Spark [9]. The pipeline has two transformations (`flatMap` and `map`) followed by an action (`reduceByKey`). In the `flatMap` transformation, each line of text is split into an array of words. Each word is then `map`ped to a *(word, 1)* key-value pair. Finally in the `reduceByKey` action, all key-value pairs are merged by adding the values of pairs with the same key. Comparing this to the Hadoop *word-count* example in listing 2.1, the Spark code is much more concise. There is also less code devoted to configuration and the data flow can be more readily seen from the pipeline. (The Java version is slightly more verbose as the programmer needs to specify types for variables and tuples, and to use anonymous class syntax for passing functions in Java 7 and earlier.)

```
object WordCount {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("word␣count")
      val spark = new SparkContext(conf)
    val textFile = spark.textFile("hdfs://...")
    val counts = textFile.flatMap(line => line.split("␣"))
                         .map(word => (word, 1))
                         .reduceByKey(_ + _)
    counts.saveAsTextFile("hdfs://...")
    spark.stop()
  }
}
```

Listing 2.2: Spark *word-count* example in Scala.

Spark also allows programmers to trade off execution time and storage space by caching intermediate RDDs in memory or on disk. This is useful to avoid recomputing results in frequently-used RDDs.

Internally, Spark operates on small partitions of RDDs in memory at a time. Operations are implemented as MapReduce computations (some stages are optional depending on the operation; see figure 2.5 for an illustration). For example, `flatMap` and `map` correspond to map stages while `reduceByKey` implements the shuffle and reduce stages. Operations with only map stages do not require nodes to communicate with each other, unlike those with reduce stages. These tasks are submitted to executors on each node.

Spark does not force any particular input source on the programmer. Instead it supports a number of them ranging from simple text files to distributed filesystems and services (HDFS and AmazonS3). This makes Spark compatible with existing storage services and thus speeds up its adoption rate. The downside is that the distribution of data is outside of Spark's control, limiting its ability to optimise data flows.

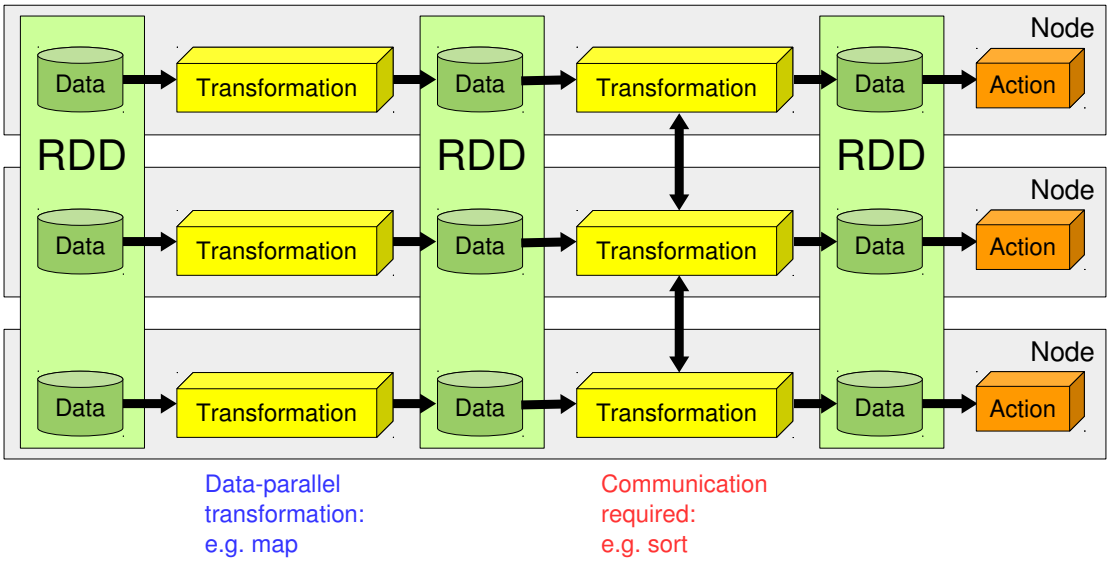


Figure 2.5: Spark pipeline showing the different types of operations (two transformations and an action).

Simple Scalable Streaming System (S4) and Storm

S4's programming model differs significantly from Spark as it is an event processor for unbounded data streams [61]. It defines a *stream* as a sequence of key-value tuples, which are selectively consumed and processed by *Processing Elements* (PEs), which optionally emit more key-value tuples. A PE processes tuples associated with a specified event and optionally containing a specified key-value pair. As a result, potentially many PEs can be created during a computation, and they are subject to garbage collection to reduce memory usage.

S4 removes the limitation of MapReduce's fixed *map-shuffle-reduce* stage sequence. However, it still requires the use of key-value pairs which limits its flexibility.

Storm is also an unbounded event processor, but does not limit the programmer to using key-value pairs. Storm executes *topologies* which run indefinitely on a cluster. A topology defines a directed acyclic graph of vertices and data *streams* (edges). A stream consists of an unbounded sequence of tuples. A vertex is either a *spout* (data source – a Twitter feed, for example) or a *bolt* (consumes and processes streams, and may emit new streams, similar to a PE; see figure 2.6 for an illustration). This is also an improvement over S4 as it includes data sources (spouts) in the model and uses a static topology thus removing the need to garbage collect bolts.

To help map a Storm topology onto a cluster, each bolt is partitioned into tasks, with each node running at least one task. Programmers specify a *stream grouping* for each stream, which determines how tuples in the stream are forwarded to tasks in the destination bolt. For example:

- A *shuffle* grouping forwards tuples to a random task.
- A *fields* grouping forwards tuples based on field values, with those having the same values going to the same task. This is conceptually similar to the MapReduce *shuffle*

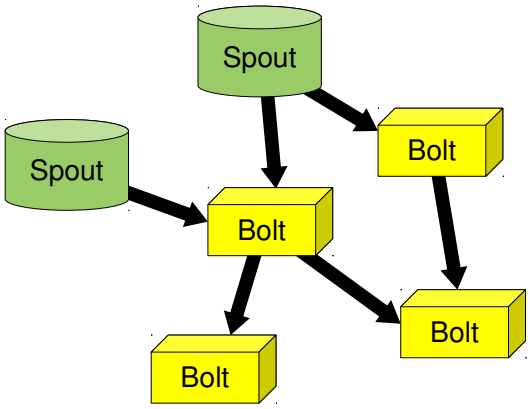


Figure 2.6: A Storm topology. Arrows indicate data flow.

stage.

- A *local or shuffle* grouping forwards tuples to a local task. If there are none, a random remote task receives them (as in a *shuffle* grouping).

Both S4 and Storm are implemented using the actor paradigm, which contributes to the frameworks being easily programmable.

2.5.3 High-level languages

This section briefly reviews a number of frameworks that provide high-level languages geared towards processing large datasets. Hive, Shark and Pig cater to requirements for processing large relational datasets.

Hive and Shark

The Apache Hive framework provides HiveQL, a SQL-like language, which is translated and executed as Hadoop jobs. Apache Shark is a related framework that maps HiveQL to Spark jobs instead. Data is first loaded into Hive tables with the `LOAD` statement, and can subsequently be used and manipulated with `SELECT`, `INSERT`, `UPDATE` and `DELETE` statements. Data in Hive tables can be written back to disk with the `INSERT OVERWRITE DIRECTORY` statement. Hive tables are stored in HDFS. HiveQL also allows Hive tables to be partitioned according to specified column values, which potentially avoids time-consuming full-table scans. Although this frees the programmer from low-level implementation details, not all datasets fit into the relational data model.

Pig

The Apache Pig framework introduces a data-parallel language called Pig Latin for analysing relational data. Like Hive, data first needs to be loaded into relations before it can be analysed. However, unlike Hive, Pig Latin is a procedural language and provides a high-level alternative to declarative languages (primarily SQL and its derivatives), where the flow of data is not readily apparent in queries. This also gives programmers more control over data flows instead of having to rely on query optimisers.

2.5.4 Extending existing models, frameworks and languages

Twister, HaLoop and ReStore

Twister [28] extends the MapReduce programming model by adding iteration and caching of data across MapReduce jobs. This reduces the amount of data read from disk during each iteration and thus speeds up execution. HaLoop [18] modifies Hadoop to make it easier to build iterative applications, and also aggressively caches loop-invariant data to reduce recalculation. Although better support for iterative MapReduce improves readability of job submission code, it does not address the low level nature of the programming model.

ReStore [29] is an extension to Pig which attempts to store and reuse intermediate results of jobs and sub-jobs, making it possible to speed up programs without any code modification. Decisions to store and evict results are rule- and heuristic-based.

CoHadoop and CARTILAGE

CoHadoop [30] proposes a lightweight extension to Hadoop to allow control over the placement of data. This follows from the observation that an even partitioning of data over a cluster (which HDFS tries to achieve) is not always as efficient as a custom partitioning. It introduces for each file an optional *locator* property that is used as a hint for placement in the cluster.

Similarly, CARTILAGE [45] recognises that suboptimal data layout is often a cause of poor performance in HDFS-based systems and so allows more user control over data partitioning, replication and layout.

EARL

The EARL framework [48] extends MapReduce to handle early and incremental calculations. EARL initially operates on a small subset of the input data, and estimates the error of the produced result. If the error is too great, then more data is brought in and the result refined. This is useful if one does not require an exact result and thus does not need to process the entire dataset.

Ricardo

Ricardo [24] integrates the statistical modelling language R with Hadoop through a JSON-based query language (JAQL) [17] to allow use of R on large datasets. JAQL queries embedded in R are executed in parallel on the cluster.

Parallelization Contracts

As discussed in section 2.5.1, MapReduce allows programmers to specify *map* and *reduce* functions which are executed in fixed *map-shuffle-reduce* stages. The Parallelization Contract (PACT) model [16] generalises MapReduce to support more user-defined functions and acyclic data flows. In PACT, programmers specify input contracts that determine how data is partitioned before being processed by user-defined functions. For example,

the *map* contract partitions a set of key-value pairs into singleton sets and passes each set independently to the mapping function. The *reduce* contract groups key-value pairs by their keys and passes each set of pairs to the reduction function. PACT also defines a number of input contracts that accept multiple input sets. Optionally, to aid compiler optimisation, output contracts that guarantee certain properties of processed data can also be specified.

2.5.5 Improving the performance of existing frameworks

Besides enlarging the cluster, there are other ways to speed up Big Data computations.

Using faster hardware

Many Big Data computations are I/O-bound and thus may benefit from faster hardware at bottlenecks [69]. For datasets that are distributed over the whole cluster, performance can be improved by using solid state disks (SSDs) [44, 81] and faster network interconnects such as 10 gigabit Ethernet and Infiniband [81].

Faster network speeds also enable new possibilities for mapping existing programming models to the underlying architecture. There have been attempts to use Lustre (a high-performance and POSIX-compliant distributed file system) as the storage layer for Hadoop [75] to avoid loading datasets into HDFS and for applications that require files to be overwritable. This is a departure from the traditional assumption that data locality is important.

Tachyon

Tachyon is an in-memory data storage layer that is compatible with many popular Big Data frameworks [52]. It sits between the framework (such as Hadoop or Spark) and the underlying data storage (such as HDFS), caching datasets in memory to speed up data access. For datasets that do not fit in memory, it also supports using SSDs for caching. The advantage of using Tachyon is that no major changes to applications is needed.

2.6 Java 8

The most popular open-source Big Data frameworks are targeted towards Java and/or higher-level languages that run on the JVM (eg. Scala). This is mainly because of the write-once-run-anywhere nature of JVM bytecode making it easier to deploy code on heterogeneous clusters. Another contributing factor is the optimised state of current JVMs (eg. Oracle and OpenJDK) narrowing the gap between machine code and bytecode execution times [80]. This reduces the need for programming Big Data systems – which are potentially complex – in a non-portable and low-level language.

The recent introduction of Java 8 has brought the language a step closer to having native Big Data capabilities due to the addition of the following:

- Java 8 Streams (specified in JEP 107) [66] provide a new programming model for data parallel computation on a single machine. The motivation of introducing Java

8 Streams was for programs to scale to multicore systems with little or no modification [66]. On such systems it is able to parallelise a computation over multiple threads to speed up execution.

- Lambda expressions (specified in JSR 335) [67] simplify functional programming in Java, allowing concise instantiation of functional interfaces instead of using the anonymous class syntax. These are used extensively in Java 8 Streams, making its code more readable. Lambda expressions were inspired by functional programming languages such as Haskell [40]. They were also introduced to keep up with the JVM languages Scala [27] and Clojure [73], both of which support functional programming.

2.6.1 Lambda expressions

A lambda can be specified in place of a value whose type is a *functional interface* – an interface with only one abstract method. For example, a thread which was originally declared using anonymous class syntax:

```
Thread th = new Thread(new Runnable()
{
  public void run()
  {
    ...
  }
});
```

can instead be written more easily with a lambda:

```
Thread th = new Thread(() ->
{
  ...
});
```

Lambdas have a flexible syntax. For example, an expression can be used in place of a single return statement, allowing for more concise expression of one-line functions. Parameter types can also be omitted, as they will be inferred from by the compiler. The parentheses surrounding the parameters can also be omitted if there is one parameter. Thus, for a lambda that doubles an integer, the following representations are equivalent:

```
(int x) -> { return x * 2; }
(int x) -> x * 2
(x) -> x * 2
x -> x * 2
```

The return type of a lambda is not stated in the syntax, but return values have to match the corresponding abstract method's return type. Thus, if the required return type is `int`, a lambda such as:

```
(x, y) -> { if (x == y) return x + y; }
```

is illegal and will generate a compile error because there is no return value for the case `(x != y)`.

In the context of parallel programming, lambdas do not expose data parallelism directly, but they make it easier for libraries to implement it. Consider the following sequential loop that operates on all elements in an iterable:

```
for (Shape s: shapes)
  s.type = CIRCLE;
```

Libraries are unable to parallelise this, as it will be compiled into a sequential loop by most compilers. With lambdas and a library-defined `forEach` method that executes a given lambda in parallel, it can be rewritten as:

```
shapes.forEach(s -> { s.type = CIRCLE; });
```

There are several limitations to what lambdas in Java 8 can do:

- Variables or fields that are outside the lambda's scope can be accessed, but cannot be modified, from inside the lambda. Attempts to modify these values will generate a compilation error.
- When parallelising a sequential loop, it is possible to skip an iteration – normally done with the continue statement – with a return statement. However, it is more difficult to end the loop early because subsequent iterations after the desired end-point may potentially be executing or have been executed. Some form of co-ordination between individual lambdas may be required in this case.

2.6.2 Streams

A Java 8 *stream* is a sequence of data elements that can be processed by a *pipeline* of operations. Streams can be generated from several sources, including:

- Collections, by calling the `stream` method if the desired stream should be sequential, or the `parallelStream` method for a parallel stream;
- Arrays, by calling the `Arrays.stream` method;
- Factory methods in the `Stream` class;
- Files, by calling the `BufferedReader.lines` method.

After retrieving the stream from a source, a pipeline of aggregate operations (formerly called bulk data operations) can be performed on it, consisting of zero or more intermediate operations followed by a terminal operation. From the programmer's perspective, an intermediate operation returns a new stream of processed elements from the given stream, and a terminal operation returns a non-stream result.

Streams, pipelines and operations have the following properties and restrictions:

- Pipelines are evaluated lazily. The terminal operation triggers computation of the pipeline. Execution of the pipeline is thus deferred until the terminal operation is called.

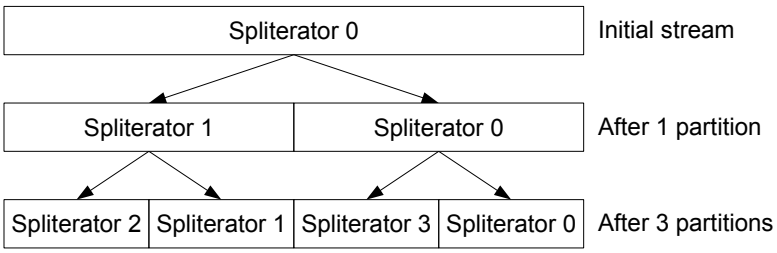


Figure 2.7: Recursive partitioning using spliterators.

- Pipelines are short-circuiting. Only enough elements are consumed by the pipeline as required. This allows unbounded datasets (such as those generated from infinite sequences) to be used as data sources. Operations that cause this behaviour in pipelines are called short-circuiting operations. Examples of such operations are `findFirst` and `limit`.

- Pipelines are linear. There is a single stream source, and there is no branching mechanism for routing elements to different downstream operations.

- Streams can be traversed at most once. To use the data source again, a new stream has to be created.

- Operations should not change the data source if the source does not support concurrent modification.

Java 8 Streams further classify intermediate operations as stateless and stateful, depending on whether the operation needs to hold any state as data passes through. For example, `map` is a stateless operation as each element can be processed independently of another. However, the `distinct` operation (remove all duplicate elements from the Stream) is stateful because it must keep track of all encountered elements.

Details of the Java 8 Stream API can be found at [68]. The rest of this section gives details on how Streams are implemented.

Parallel execution of pipelines

Execution of a sequential pipeline is achieved by iterating over all its elements. While an iterator is sufficient for a sequential pipeline, a *spliterator* is needed to operate on a parallel pipeline. A spliterator recursively partitions the stream by "splitting" itself to create child spliterators, allowing threads to then traverse the multiple spliterators in parallel.

To execute a pipeline in parallel, a spliterator covering the entire associated stream is first created. Recursive partitioning is achieved by creating child spliterators from parent spliterators as shown in figure 2.7. During each partition, a thread is given access to a spliterator and attempts to partition it using the `Spliterator.trySplit` method. If the spliterator is larger than a threshold specified in its implementation, a new spliterator covering part of it is created and submitted for further partitioning by other threads, while the current thread is left with the remainder of the spliterator.

```
import java.nio.file.*;
import java.util.*;
import java.util.regex.*;
import java.util.stream.*;

public class WordCount
{
    public static long wordcount(Stream<String> lines)
    {
        Pattern delim = Pattern.compile("\\s+");
        return lines
            .flatMap(line -> Stream.of(delim.split(line)))
            .count();
    }

    public static void main(String[] args)
    {
        Path filename = Paths.get(args[0]);
        Stream<String> s = Files
            .lines(filename)
            .parallel();
        System.out.println(wordcount(s));
    }
}
```

Listing 2.3: Simple word-count application using Java 8 Streams.

The fork-join framework

Parallel streams are executed by Java's common fork-join thread pool [64]. Introduced in Java 7, the fork-join framework is a type of Executor framework that allows programmers to specify tasks that can be subdivided and executed in parallel on multicore systems. Such tasks are submitted to a fork-join pool, which consists of a set of worker threads. Each worker thread has a task queue, and when empty, steals tasks from other threads' queues. To simplify usage of the framework, a common fork-join pool is defined, with the number of worker threads defaulting to one less than the number of cores on the system. This common pool receives tasks from executing pipelines. In the context of Java Streams, a task is the pipeline that operates on part of the stream covered by a spliterator.

The behaviour of the fork-join framework gives rise to a partial ordering of splits. Thus (referring to figure 2.7), the partition that creates Spliterator 2 may happen before or after the partition that creates Spliterator 3, but both may not happen before the partition that created Spliterator 1.

2.6.3 Word-count example

To illustrate the Java 8 Stream programming model, listing 2.3 shows the the word-count example implemented in Java 8 Streams.

Firstly, in the `main` method, `Files.lines` returns a sequential stream of lines (strings) from the file pointed to by `filename`. Next, the call to `parallel` marks the stream as a parallel stream, allowing it to be partitioned and the pipeline to be executed in parallel.

The actual pipeline consists of two operations:

- The `flatMap` operation splits each line into words (Java `String`s) which are then

passed downstream as individual data elements.

- The `count` operation then counts the number of data elements encountered and returns the result in a long integer.

2.7 Summary

This chapter has reviewed several parallel programming models, Big Data programming models and frameworks, as well as introduced the new stream-based programming model that comes with Java 8. A summary of characteristics for the reviewed Big Data programming models as well as for the Java 8 Stream model is given in table 2.1.

In general, the main differences between conventional parallel programming models and Big Data programming models are:

- Big Data models are high-level compared to conventional models, but they restrict the progammer more. For example, conventional models support task parallelism while Big Data models have very limited or no support for task parallelism and only support data parallelism. This is due to Big Data models prioritising the locality of data. They are designed to keep data local as far as possible, while conventional models are for general parallelism and do not place the same restrictions on the programmer. The Big Data models handle most of the splitting, communication and mapping of tasks implicitly while more of this work is given to the programmer in conventional models.

- Fault tolerance is implemented in Big Data models used in production environments but is usually lacking in conventional models [85]. This is because support for large clusters is needed in Big Data models as the likelihood of a node failure increases as the size of a cluster grows, thus fault tolerance is needed in production-quality Big Data models.

Of the Big Data programming models reviewed, MapReduce was the earliest model developed, and was popularised by Hadoop. Hadoop was successful in addressing the Big Data issues mentioned in chapter 1:

- Data is distributed over the cluster by the file system (HDFS).

- Computation is split into three stages: map, shuffle and reduce.

- Data is read from HDFS into mappers (locally as far as possible), shuffled to the appropriate reducers, and the results are written back to HDFS.

Hadoop was simple to program but low-level (for Big Data models) and restrictive. It limited programmers to implementing mappers and reducers, required them to think in terms of key-value pairs, as well as forced datasets to be stored in HDFS. To work around the limitations, there have been attempts to extend Hadoop and MapReduce (to support iterative computations or to further generalise MapReduce, for example). Less restrictive programming models were also developed. Some models allow programmers to specify graphs of computations (Storm and S4). Others, such as Spark, use pipelines to describe

Model(s)	Type	Underlying paradigm or model	Parallelism	Attributes
MPI	Framework	SPMD	Task, data	Fragmented, message-passing
OpenMP	Framework	Shared memory	Task, data	Global-view
Chapel, X10	Language (imperative)	APGAS	Task, data	Global-view
Hadoop and extensions	Framework	Task farming	Data	Global-view, mapper/reducer in separate classes
Twister	Framework	Task farming	Data	Global-view, mapper/reducer in separate classes
Spark	Framework	Task farming	Data	Global-view, pipeline
Storm	Framework	Actor	Data	Global-view, computation graph, bolt code in separate classes
S4	Framework	Actor	Data	Global-view, computation graph, PE code in separate classes
Hive, Shark	Language (declarative)	Hadoop	Data	Global-view, SQL-like query
Pig	Language (imperative)	Hadoop	Data	Global-view
PACT	Framework	MapReduce (generalised)	Data	Global-view, computation graph
Java 8 Streams	Framework	Shared memory	Data	Global-view, pipeline

Table 2.1: Summary of programming model and framework characteristics.

computations, greatly improving code readability. These newer models tend to support multiple data source formats to maximise adoption of the models and to operate with existing data. A third approach was to develop high-level languages on top of Hadoop and MapReduce. Some of the languages, such as Pig, are new while others (Hive and Shark) resemble SQL. With faster hardware to store or move data, new ways to map existing programming models to architectures emerged.

The next chapter explores the feasibility of using Java 8 Streams for Big Data processing.

Chapter 3

Using Java 8 for Big Data

The previous chapter introduced several Big Data programming models as well as the Java 8 Stream model. In this chapter:

- Section 3.1 identifies the attributes of Big Data programming models.
- Section 3.2 discusses the Java 8 Stream model with respect to these Big Data attributes.
- Section 3.3 lists the requirements of an extended, distributed stream model based on Java 8 Streams.
- Finally, section 3.4 summarises this chapter.

3.1 Attributes of Big Data programming models

From the Big Data programming models and frameworks reviewed in section 2.5, several attributes can be identified.

3.1.1 Architecture independence

Many of the popular Big Data frameworks are written in and have bindings for architecture-independent languages such as Java, Scala (both which use the JVM) and Python. They can run on any supported OS without having to re-compile the binaries. This allows clusters to be built from different OSes and hardware architectures.

3.1.2 Ability to process very large datasets

One of the most important attributes of Big Data programming models is the efficient and transparent processing of very large datasets. This implies some form of data distribution mechanism as these datasets often do not fit in a single machine. For example, Hadoop uses its HDFS file system to transparently partition large files into blocks and spread them across the cluster. It also handles cases when data items straddle multiple blocks. Thus, the programmer can code without knowledge of blocks and other HDFS internals. Spark, on the other hand, has wider support for data sources from text files on the local filesystem to distributed filesystems [13] (including HDFS, Cassandra [14] and Amazon

S3 [3], many of which are fairly mature and have built-in fault tolerance and replication). Finally, Twister by default treats a dataset as a group of files on each node with the same pathname on the local filesystem [28]. Data is also not allowed to straddle multiple files.

3.1.3 Support for different data formats

With the proliferation of Big Data frameworks, recent frameworks began to emphasise extensibility and interoperability with parts of older frameworks, especially of datasets. Besides giving users more choices, this benefits adopters of early frameworks who already have large volumes of data stored in existing Big Data formats, allowing them to experiment with newer frameworks more easily. For example, Hadoop introduced the HDFS file system (reviewed in section 2.5.1), which can be used as a data source in newer frameworks such as Spark and Storm. Data sources that are not already supported can be added without modifying the framework's internals. A more extreme example is Tachyon (reviewed in section 2.5.5), which supports many different frameworks.

3.1.4 Data locality

Data locality is also an important Big Data attribute. Since it is expensive to move data between nodes in a cluster, data needs to be kept local as much as possible. Nevertheless, Big Data programming models also provide a mechanism for moving data for cases where this is necessary (in data aggregation, for example). Hadoop, which implements the MapReduce programming model, has a shuffle stage where all key-value pairs emitted from mappers are sent to the appropriate reducer depending on the key. In Spark, the movement of data (if any) is determined by the transformations and actions specified in the pipeline. For example, the `sortByKey` transformation may move data across nodes to matching key destinations, while the `map` transformation keeps data local. In Storm, the movement of data is determined by the stream groupings specified in the topology. For example, a *local or shuffle* grouping maintains locality of data between bolts as far as possible, while a *fields* grouping likely involves movement of data across nodes.

In most cases (as above), data locality is implicit in the programming models and transparent to the programmer. There are also extensions to existing models (reviewed in section 2.5.4) that give programmers more control over where data is placed.

3.1.5 Transparent fault tolerance

The programmer normally does not need to be concerned with fault tolerance in Big Data frameworks, as it is handled transparently by the programming model. Unlike high-performance computing, this is important for Big Data because applications are deployed over large numbers of unreliable nodes. For example, Hadoop uses heartbeats to detect a failed node and automatically reruns jobs that were assigned to it on other nodes.

Due to time constraints, this thesis does not address fault tolerance. However, the transparent nature means that fault tolerance can be added in the future without significantly affecting the programming model.

3.1.6 Functional programming

Some Big Data programming models use functional programming to simplify data-parallel programming. For example, Spark's pipelines are more easily written in Scala due to its functional programming syntax than in Java 7.

3.2 The Java 8 programming model and Big Data attributes

Although there are several advantages in using Java 8 for Big Data computing, there are also issues with its programming model. This section reviews Java 8's ability to handle Big Data.

3.2.1 Architecture independence

One of the main advantages of Java is its *write-once-run-anywhere* paradigm. The JVM is supported on major OSes, and has also been optimised on popular hardware architectures (such as x86, x86-64, ARM and PowerPC).

3.2.2 Ability to process very large datasets

Java 8 Streams are unable to handle very large (and distributed) datasets because they only operate within a JVM and there is no concept of a cluster. Furthermore, it does not support on-disk datasets well. These deficiencies are elaborated below, and are addressed in chapters 4 and 5.

No concept of a cluster

Java 8 Streams exist within a single JVM, and JVMs tend to only support individual SMP or ccNUMA machines, therefore stream computations will be limited to individual machines. (Since there are no distributed JVMs in wide use, this thesis assumes that a JVM does not span multiple machines or nodes.) There is no concept of a cluster and hence of distributing data or computations to other nodes in a cluster, thus on its own it does not have the ability to process very large datasets (as mentioned in section 3.1.2). Although Java supports RMI which helps in writing distributed applications, there is no integration between RMI and Java 8 Streams.

A difficulty of large-scale computation is that it requires support for both distribution of computation and distribution of data. Both methods potentially speed up processing, depending on the workload. An I/O-bound workload will see a speed improvement when distributing data, whilst a CPU-bound workload will benefit more from distributed computation.

Since a standard JVM has a single heap and runs on a single node in a cluster, multiple JVMs are needed for distributed, cluster computing. Thus, the computation has to be partitioned across all JVM heaps and memory access on a remote heap must be done explicitly with the use of middleware such as MPI. However, simulating a single cluster-wide heap makes the cluster more tightly coupled and thus increases the difficulty of implementing fault tolerance.

Java 8 Streams support short-circuiting (reviewed in section 2.6.2). However, using short-circuiting in a distributed computing model may have performance implications. Since the current model supports computation only in a single node, each thread in the fork-join pool works a single data element at a time through the pipeline. Thus, the decision to retrieve the next element is known immediately after the last element is processed by the terminal operation (or if another thread signals completion) – that is, no waiting is necessary. When a pipeline is distributed across a cluster, the terminal operation will need to wait for existing data items on other nodes to be processed before knowing if more data needs to be processed. If the data source is also distributed, there is the additional overhead of deciding which node to get more data from. These pieces of information will have to be communicated across nodes, with increased latency as a result. Hence, the short-circuiting feature needs to be modified when applied to a cluster.

No support for very large, potentially distributed data sources

Only in-memory datasets are well supported (by the Java collections framework), but these are not useful for very large datasets as it requires loading the entire dataset into memory. Datasets from existing on-disk sources (eg. `BufferedReader.lines`) are not optimised for large amounts of data and are likely to cause out-of-memory errors with large files. Furthermore, there is no concept of a dataset that is distributed over a cluster.

3.2.3 Support for different data formats

There is currently no support for Big Data formats as Java 8 does not address Big Data issues. However, the Java collections framework is extensible and can be used as a base for these datasets. Chapters 4 and 5 describe how collections can be extended to support Big Data processing and thus the various Big Data formats.

3.2.4 Data locality

There was no previous research on data locality in Java 8 Streams as it was recently introduced. Chapter 4 describes experiments that were done to determine the data locality of Java 8 Streams. Chapter 5 extends data locality to a distributed environment.

3.2.5 Fault tolerance

Since Java runs on a single JVM, it has no concept of cluster-based fault tolerance. Although RMI allows distributed programs to be written, it is not integrated with Java 8 Streams and is not fault-tolerant by default, so programmers have to implement their own.

3.2.6 Functional programming

To facilitate functional programming, lambda expressions have been introduced in Java 8. Although they have limitations (discussed in section 2.6.1), they are still useful, especially for specifying pipelines in Java 8 Streams.

3.3 Requirements

The last section brought up issues faced by using Java 8 Streams for Big Data. From these, a set of requirements for an extended stream framework is derived:

1. **It must support running on a cluster of nodes.** This is to allow data-parallel computation to take place on a group of computers built from commodity hardware, as opposed to more expensive high-performance hardware.

2. **It must support large datasets, both on a single node and distributed across nodes.** Distributed datasets scale with the number of nodes in the cluster, and can potentially improve data locality of a computation.

3. **It must maintain data locality within a node and across nodes.** This makes computation efficient within each node and within the cluster.

4. **It must have operations which distribute data and computation across nodes.** This is to allow for cases where data needs to be explicitly transferred between nodes.

5. **It must be a drop-in replacement for Java 8 Streams.** Backward compatibility is not a priority in Big Data frameworks given that their development is fast-paced and that this area of research is still at the cutting edge. However, making the new model backward compatible with the current Java 8 Stream model can minimise porting effort and even speed up adoption of the new model. For example, a pipeline for filtering and counting data items (see listing 3.1) would remain the same regardless of the data source used.

```
long countPositive(IntStream s)
{
  return s
    .filter(n -> n > 0)
    .count();
}

void existingMethod()
{
  int[] numbers = { ... }; // local data source
  IntStream s = Arrays.stream(numbers);
  long positives = countPositive(s);
}

void newMethod()
{
  IntStream s = distributedSourceOfInts.stream(); // distributed data source
  long positives = countPositive(s);
}
```

Listing 3.1: Example code showing how the new programming model can maintain backward compatibility with Java 8 Streams. The pipeline in the `countPositive` method filters out negative integers and zeros, and counts the remaining positive integers. Its implementation stays fixed while the data sources change (from a local source in `existingMethod` to a distributed one in `newMethod`).

3.4 Summary

This chapter has analysed the desirable attributes of Big Data programming models and compared them with what is provided by Java 8 Streams. Having derived the list of requirements in section 3.3, the next two chapters describe extensions to the Java 8 Stream model that fulfill them. Chapter 4 focuses on single-node extensions and chapter 5 builds on them to propose a fully distributed solution.

Chapter 4

Supporting Big Data on a Single Node

The previous chapter (specifically section 3.2.2) has highlighted problems with Java 8 Streams in using existing data sources within a single node. Therefore, as a start, a new data source that overcomes these limitations on a single node is introduced in section 4.1. In section 4.2, a single-node performance comparison of Hadoop and Java 8 Streams (backed by existing and new data sources) was carried out. Additionally, section 4.2 compares the data locality of Stored Collection-backed and array-backed Java 8 Streams. Finally, section 4.4 concludes the evaluation and summarises the chapter.

4.1 Stored Collections

One of the data sources supported by Java 8 Streams is the (in-memory) collection. This is part of Java's collection framework, and the built-in implementations store their data items in the Java heap. However, being in-memory is a major drawback. It implies populating the entire collection before any operations can be performed, resulting in a potentially long delay while this takes place. Furthermore, heap memory is usually small compared to disk space, so for Big Data computations, there may not be enough heap memory to load the entire dataset from disk. Hence, the built-in collections are unsuitable for Big Data processing.

To extend its functionality, the idea of a *Stored Collection* is introduced. A Stored Collection allows data to be read from a file on-demand, thus eliminating the initial population step and the memory issues that accompany it.

Stored Collections allow arbitrarily large files on a single node to be used as data sources without any modification of the Java 8 Stream framework. It does so in batches of constant size, so as not to use up too much memory at any time (unlike the stream created by the `BufferedReader.lines` method, which has too large a maximum batch size). They are also designed to be extensible, so other on-disk data sources (such as results from database queries) can be used as Stored Collections by extending the `StoredCollection` and `Spliterator` classes (see listing 4.1).

For example, to implement a `SQLStoredCollection` that contains rows from executing a query in a database:

- It should extend the `StoredCollection` class.
- A constructor which accepts the necessary information to open and read from the database (e.g. the filename and query string) should be defined.
- The `spliterator` method should return a new `SQLStoredSpliterator` object.

The corresponding `SQLStoredSpliterator` class should extend the `Spliterator` class and implement the following key methods:

- A constructor which accepts the necessary information to open and read from the database (e.g. the filename and query string). In many cases, the parameters will be similar to those in the Stored Collection constructor.
- The `tryAdvance` (and optionally the `forEachRemaining` and `trySplit`) method. The `trySplit` method attempts to split part of the dataset into another spliterator to allow for concurrent execution (defaults to sequential execution if not overridden). The `tryAdvance` method reads a single item in the range covered by the spliterator. The `forEachRemaining` method reads all remaining unread items in the range covered by the spliterator (defaults to repeatedly calling the `tryAdvance` method if not overridden).

Stored Collections are backward compatible with read-only in-memory collections and, optionally, writable in-memory collections as sources in Stream computations (see listing 4.2 for an example). They replace instances of creating and populating an in-memory collection with creating a `StoredCollection` object. If data needs to be written back to disk, one of two approaches can be used:

- If the underlying data format supports modifying data items (e.g. SQL databases), the Stored Collection's `add` and (optionally) `addAll` methods can be overridden to appropriately modify the dataset.
- If the underlying data format does not allow modifying data items once written (e.g. HDFS), the above-mentioned approach may be used with the `add` and `addAll` methods throwing an exception if attempting to modify an existing dataset.
- If the underlying data format does not allow modifying data items once written, a separate Stored Collection that only writes to newly created datasets can be implemented. This thesis uses the latter approach.

4.2 Implementation and evaluation of Stored Collections

To evaluate the performance of Stored Collections, a `StringStoredCollection` was implemented and used as data sources for Java 8 Streams. The corresponding `StringStoredSpliterator` was implemented to partition the file into byte ranges and, to resume traversal after a partition, an algorithm to retrieve the next full line was used. For simplicity, a partition generates a spliterator spanning approximately half the size of the current one, and no more partitions are made if the resulting spliterator is smaller than 1MB in size.

```
package dstream.util;

import java.util.*;

public abstract class StoredCollection<E> extends AbstractCollection<E>
{
  @Override
  public int size()
  {
    // Default implementation scans entire dataset
    // Override for a faster implementation
    long n = parallelStream().count();
    return (int) (n < Integer.MAX_VALUE ? n : Integer.MAX_VALUE);
  }

  @Override
  public Iterator<E> iterator()
  {
    // Uses the spliterator implementation but does not parallelise
    return new IteratorFromSpliterator<E>(spliterator());
  }
}
```

Listing 4.1: The `StoredCollection` abstract class.

```
public void main(String[] args)
{
  // Create Stored Collection of lines from file
  String filename = "file.txt";
  Collection<String> lines = new StringStoredCollection(filename);
  // Compute number of lines
  long numLines = lines
    .parallelStream()
    .count();
  System.println("Line␣count:␣" + numLines);
  // Compute number of empty lines
  long numEmpty = lines
    .parallelStream()
    .filter(line -> line.length() == 0)
    .count();
  System.println("Empty␣line␣count:␣" + numEmpty);
}
```

Listing 4.2: Example usage of a Stored Collection of lines: counting lines.

4.2.1 Experimental setup

Programs written for the following frameworks were compared:

- Hadoop (running on a single node).
- Java 8 Streams backed by in-memory collections (implemented as `ArrayLists`).
- Java 8 Streams backed by Stored Collections.

For each framework, two applications were run:

- *Word-count* (detailed) – outputs all encountered words and the number of occurrences of each word.
- *Grep* – outputs all occurrences of a given regular expression. The regular expression used in all tests is "`\w*z\w*z\w*`".

The machine used in all tests is a 4-socket AMD Opteron with a total of 16 cores clocked at 2GHz with 16GB of main memory and 2GB of swap, running Ubuntu 13.04. Storage consists of a single (rotational) disk. For Hadoop, version 1.2.1 on Java 7 is used with the default block size (128MB), no HDFS replication and a maximum of 16 map tasks and 16 reduce tasks. For non-Hadoop programs, a Java 8 build with lambdas (revision `h4962-20130630-b97-b00`) is used. In all cases, the default garbage collector for each Java version is used, and the disk cache is cleared before each run.

Each program is fed with plain text input data [70] with sizes starting from approximately 7.5MB. Larger input data sizes are derived by appending the original data multiple times. Each application is run 20 times for every input size.

4.2.2 Results

The results of these runs are displayed in tables 4.1 and 4.2, which contain the execution times of each application type under the default Java environment. The running times of in-memory collection programs is significantly faster than those of Hadoop. However, they do not scale well to large datasets and fail with an out-of-memory error for larger input data sizes, as the data is not able to fit into heap memory. Increasing the maximum heap size does not improve matters much, as there is still not enough main memory and data therefore spills over into swap.

For Stored Collection programs, their execution times are up to 1.44× faster and their heap usage is 2.35%–84.1% of those for in-memory collection programs. However, their execution times with respect to Hadoop appear to be application-specific. For example, they are between 1.60× and 8.53× faster than Hadoop in the *word-count* application but, for the largest dataset tested, are 1.24× slower than Hadoop in the *grep* application.

The scalability issues of Stored Collections were investigated and the implementations of `StringStoredCollection` and `StringStoredSpliterator` were improved by allowing only one thread to perform I/O on a Stored Collection at any time. This increases the chance of sequential reads and thus improves disk read speeds. Tables 4.1 and 4.2 include execution times for the improved Stored Collection program results. Speed increases of

Data size	Hadoop	In-mem collection	Stored Collection	Stored Collection (improved)
7.5MB	35.1	4.71	5.05	5.25
75MB	69.8	9.22	8.18	6.27
750MB	80.7	43.3	30.0	29.5
7.5GB	534	(Out of memory)	232	238
75GB	4830	(Out of memory)	3010	2290

Table 4.1: Average execution times (in seconds) for the *word-count* application.

Data size	Hadoop	In-mem collection	Stored Collection	Stored Collection (improved)
7.5MB	51.3	3.58	3.04	2.71
75MB	56.9	5.71	5.18	3.31
750MB	62.0	24.2	23.6	12.2
7.5GB	292	(Out of memory)	227	100
75GB	2390	(Out of memory)	2960	1100

Table 4.2: Average execution times (in seconds) for the *grep* application.

up to 1.31× for the *word-count* application and up to 2.69× for the *grep* application were obtained. The improved Stored Collection was also more scalable than before, with execution times increasing up to 9.6× for *word-count* and up to 11× for *grep* for a 10× increase in data size.

4.3 Data locality within a node

Another evaluation was performed to compare the effects of data locality within a single Java 8 JVM on two types of data sources: arrays and Stored Collections.

To illustrate the effects that arrays and Stored Collections have on memory access, assume a hypothetical quad-core computer with cache-coherent main memory divided into two regions that have non-uniform access times (NUMA nodes), and each memory area is local to a pair of cores. (This is a "black-box" approach which has its limits, but modern systems are increasing in complexity and it may not be feasible to be aware of all the effects components have as they interact with one another. Accordingly, this methodology is unable to differentiate where in the memory hierarchy effects are observed. It instead aims to reduce the observed overheads of the memory system when observed as a whole.)

By default, threads are scheduled by the OS to execute on any core. This can lead to increased execution times if a thread is migrated to a core attached to a different memory area, causing its once-local memory accesses to subsequently be remote. To prevent this from occurring, threads can be bound to a core or set of cores.

However, affinity makes scheduling less flexible and may even negate any performance gains from locality. For example, if two threads are bound to the same core, at most one of them will be running at any time, even if there are other idle cores.

4.3.1 Memory access

For array-backed streams, suppose that a large dataset is stored in the array which spans both NUMA nodes. As a simplification, assume that the work is split into four tasks – one

for each thread (with one thread per core, as in figure 4.1) – that all workloads are even and that no threads block. When a pipeline of operations is performed in parallel, each thread may not be accessing local memory, depending on the location of the section of the array and the mapping of threads to cores. Binding threads to cores may not improve access times as the required array section may still be non-local (figure 4.2). For Stored Collections, data is read into buffers that, on ccNUMA-aware JVMs, are allocated locally to each thread (figure 4.3) on a best-effort basis. This increases the chance of fast access to a buffer by its corresponding thread. However, the pathological case is still possible if the OS or JVM subsequently migrates threads to cores such that all buffers are non-local. This can be mitigated by binding the threads to cores before buffers are allocated. Alternatively, if NUMA nodes can be identified in the hardware architecture, a looser binding of threads to NUMA nodes can be made.

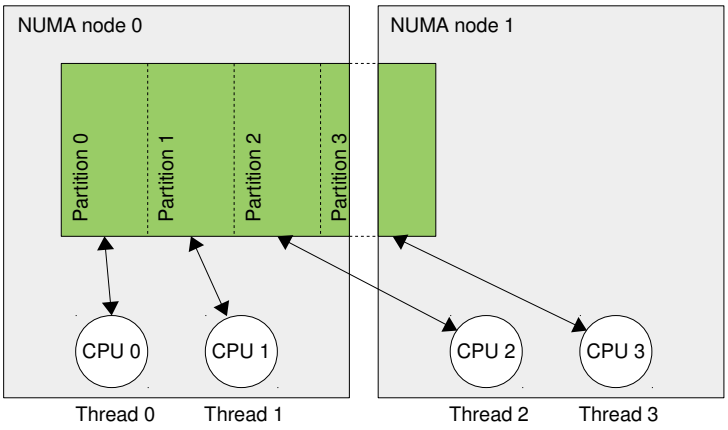


Figure 4.1: For a collection- or array-backed pipeline, threads may not be accessing local memory.

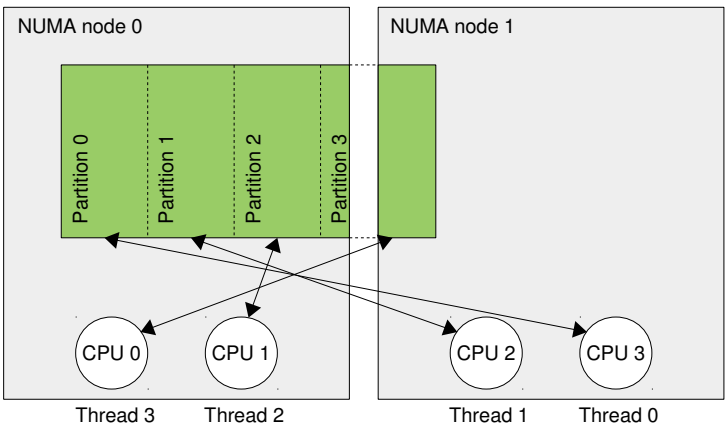


Figure 4.2: Processor affinity does not improve the performance of collection- or array-backed pipelines.

4.3.2 Disk and memory access

All data in an array or built-in collection source needs to be inserted before operations can be performed. Since sequential disk access results in fastest read throughput [44],

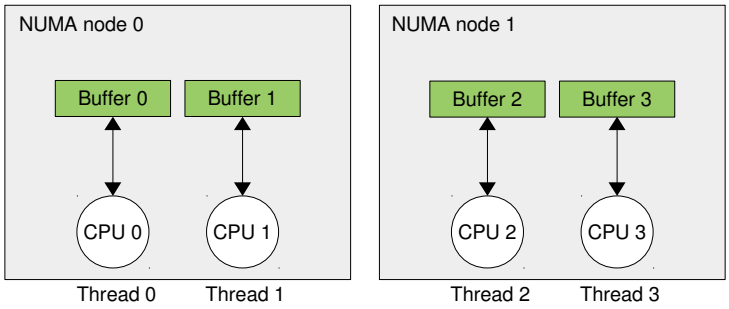


Figure 4.3: Locally-allocated buffers in Stored Collections.

a reasonable implementation would be to insert this data using a single thread. On a ccNUMA-aware JVM, the backing array would likely be local to the core that requested it, but would span memory areas if it is large. Thus for large datasets, the initial population of the array will likely involve non-local memory access. In the case of Stored Collections, data from disk is read into the buffers only when needed. The chance of local memory access is thus greater, and is further increased for bound threads. These predictions were tested by running a series of experiments described in section 4.3.3. The results obtained are given in sections 4.3.4 and 4.3.5.

4.3.3 Experimental setup and baseline

In general, CPU data caches improve the performance of programs due to temporal and spatial locality of reference – data that has been recently accessed is likely to be used again, and data that is close to those that have been accessed is likely to be used. However, for Big Data computations, disk reads take up a significant amount of time and cannot be accelerated this way. This is because the data size handled is much larger than the cache size. As a result, every attempt to access a new block of data from disk will result in a cache miss. Similarly, if the block has been accessed before, it would have been flushed out of the cache by other memory accesses in between. Hence, performance improvements in Big Data applications are more likely to come from optimising the locality of main memory access in ccNUMA architectures. Since only individual nodes in a cluster are concerned here, a machine that is representative of such nodes is considered. The machine used in all experiments is a 2GHz AMD Opteron 8350 running Ubuntu 13.04 with the following properties:

- 16 cores in total, with 4 cores per NUMA node
- 2MB of L2 cache in total, with 512KB per NUMA node
- 2MB of L3 cache shared among all NUMA nodes
- 16GB main memory in total, with 4GB per NUMA node
- Swap diabled

All programs were run with Java SE 8u5 with additional flags specifying 14GB of initial and maximum heap memory, and that the JVM use a ccNUMA-aware memory allocator.

		Allocate (CPUs)			
		0–3	4–7	8–11	12–15
Execute (CPUs)	0–3	1.304	1.371	1.352	1.508
	4–7	1.373	1.282	1.483	1.340
	8–11	1.368	1.484	1.290	1.396
	12–15	1.504	1.361	1.405	1.306

Table 4.3: Average time taken (in seconds) for a core in a specific NUMA node to sum an array of long integers allocated in a specific NUMA node.

Working on the assumption that the same thread which allocates memory will later access it, a ccNUMA-aware allocator puts memory that a thread allocates in the same NUMA node. Garbage collection is avoided during pipeline execution by reusing the SHA-256 hash objects and byte arrays that store the hash results.

The first set of experiments establish the memory access times for local and remote access separately. This is done with a C program that allocates an array of 2^{28} bytes from each NUMA node and measures the time taken for a core in each NUMA node to XOR them with a `for`-loop. Care is taken to ensure that no part of the array is initially in cache. This simulates a pass over a block of data that has just been read from disk, and the maximum difference between no-affinity and with-affinity results was obtained. Averaged results from 10 runs are in table 4.3, and indicate that the largest difference between memory accesses is about 0.23s on this machine. Percentage-wise, remote memory access is up to 18% slower than local memory access. This is a worst-case difference; the actual differences in subsequent experiments is expected to be smaller.

Subsequent experiments measured the execution times of computing the SHA-256 hashes (the SHA-256 implementation was adapted from [46]) of a stream of consecutive long integers starting from 1:

- The second set of experiments determine memory access times. Array-backed streams allocate and assign an array of long integers using the main thread and use it as the stream's source. However, Stored Collection-backed streams do not allocate this array; instead they allocate a buffer for each thread and fill it with long integers on demand. Since the JVM is ccNUMA-aware, the memory will be local to the allocating threads at the time of the requests and therefore locality of threads and data is preserved. Assignment of individual elements in the array or buffer is done sequentially and (in the case of the buffer) one thread at a time.

- The final set of experiments determine disk and memory access times. This time, the long integers are read from disk instead. Array-backed streams read them into the same array as described above, while Stored Collection-backed streams read them into the buffers on demand. This simulates a Big Data application reading and processing a dataset from disk.

For each experiment, three programs were written for each stream source:

1. Without thread affinity.

2. Binding one thread to each core.

3. Binding not more than 4 threads to each NUMA node.

All programs use 15 threads including the main thread, leaving one core free for other OS processes.

Each program is run with stream lengths of 2^{26}, 2^{27} and 2^{28} long integers, 200 times each. Results of the remaining experiments are presented sections 4.3.4 and 4.3.5. Two-tailed t-tests were performed to demonstrate that the observations are statistically significant with p-values not exceeding 10^{-10}, unless otherwise noted.

4.3.4 Memory access results

Table 4.4 shows the results obtained from 200 runs of each program. Figure 4.6 shows cumulative histograms for array-backed stream results, while figure 4.7 shows cumulative histograms for Stored Collection-backed stream results. Affinity caused a large increase in performance for array-backed streams for 2^{26} integers, while having little effect on the rest. Possible reasons for these observations were given in section 4.3, but further work is needed to fully account for the effects of affinity on streams. Nevertheless, compared to array-backed streams, Stored Collection-backed streams are still between 1.14× and 1.17× faster for the data sizes tested.

4.3.5 Disk and memory access results

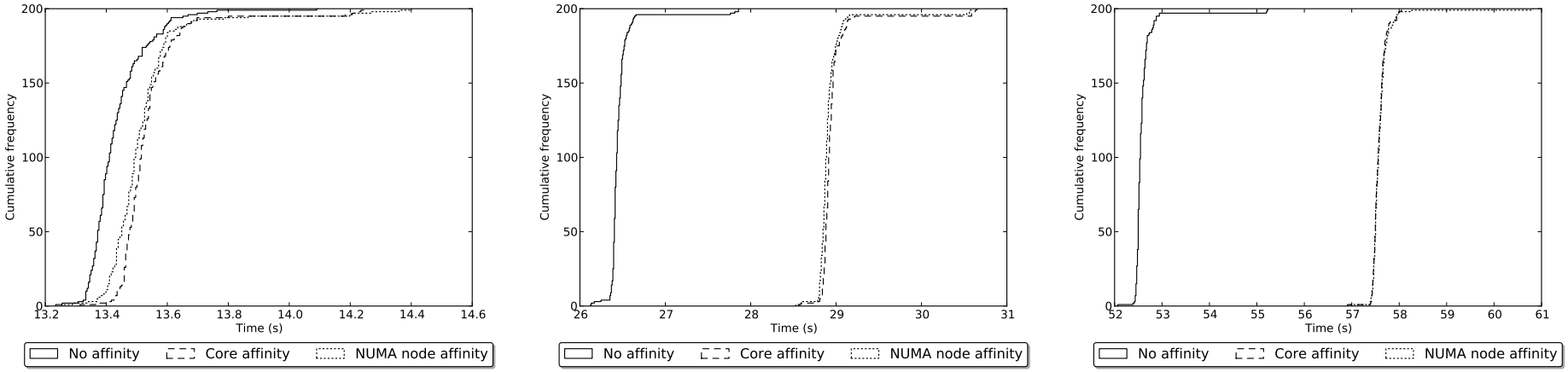
Table 4.5 shows the results obtained from 200 runs of each program. Figure 4.4 shows cumulative histograms for array-backed stream results, while figure 4.5 shows cumulative histograms for Stored Collection-backed stream results. As data sizes increase, affinity causes execution times to significantly increase for array-backed streams, but slightly decrease for Stored Collection-backed streams. Two-tailed t-tests were performed to demonstrate that the Stored Collection affinity results are statistically significant with p-values not exceeding 10^{-3}. For Stored Collection-backed streams, the differences in execution times are within the baseline results. They are also much smaller, suggesting that the OS seldom migrates worker threads, and/or that it intelligently handles thread affinity within the system. However, for array-backed streams, large increases in execution times were observed. This is likely due to data locality of the array, but more research needs to be carried out to accurately determine the causes of the large differences. Another possible reason for this is the JVM, which creates a number of background threads that perform actions such as JIT compilation, signal dispatching and garbage collection. From time to time these threads may pre-empt the worker threads, and this is likely to occur more frequently if affinities of worker threads are specified. Nevertheless, Stored Collection-backed streams are between 1.64× and 2.00× faster than array-backed streams for the data sizes tested, and are also up to an order of magnitude more predictable.

Source	Affinity	1 to 2^{26}		1 to 2^{27}		1 to 2^{28}	
		Avg time (s)	Std dev	Avg time (s)	Std dev	Avg time (s)	Std dev
Array	None	6.815	0.0979	13.165	0.221	26.074	0.474
	Core	6.743	0.0771	13.161	0.175	26.011	0.228
	NUMA node	6.725	0.0837	13.126	0.0564	26.034	0.395
Stored	None	5.864	0.0524	11.271	0.121	22.262	0.248
	Core	5.884	0.0482	11.310	0.0876	22.292	0.187
	NUMA node	5.861	0.0466	11.277	0.107	22.275	0.170

Table 4.4: Average times and standard deviations for memory access experiments (section 4.3.4).

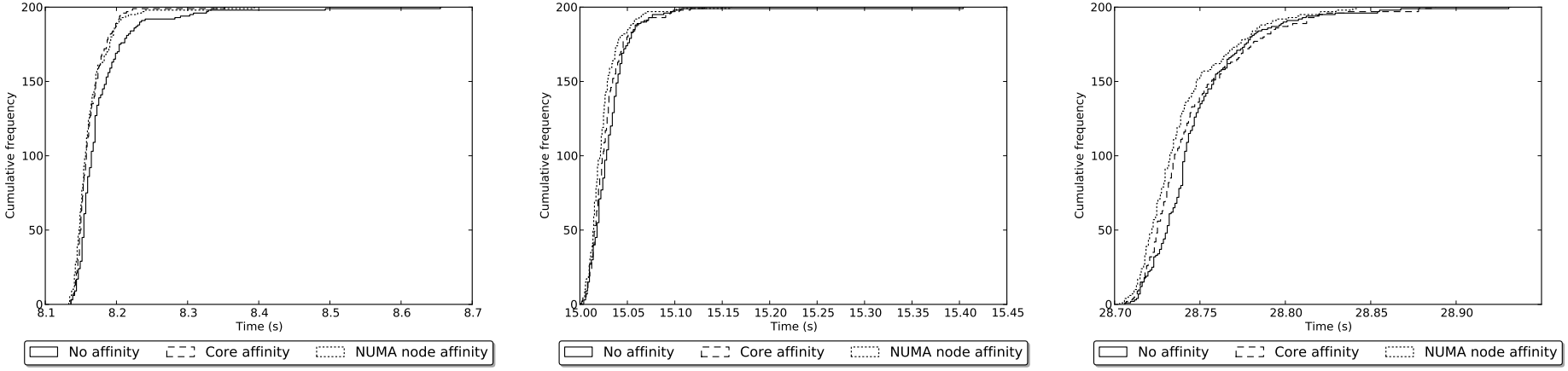
Source	Affinity	1 to 2^{26}		1 to 2^{27}		1 to 2^{28}	
		Avg time (s)	Std dev	Avg time (s)	Std dev	Avg time (s)	Std dev
Array	None	13.435	0.0954	26.472	0.206	52.610	0.340
	Core	13.541	0.126	28.974	0.275	57.580	0.131
	NUMA node	13.522	0.145	28.933	0.248	57.609	0.266
Stored	None	8.179	0.0521	15.034	0.0320	28.749	0.0295
	Core	8.164	0.0223	15.029	0.0201	28.746	0.0310
	NUMA node	8.164	0.0286	15.025	0.0184	28.740	0.0263

Table 4.5: Average times and standard deviations for disk and memory access experiments (section 4.3.5).



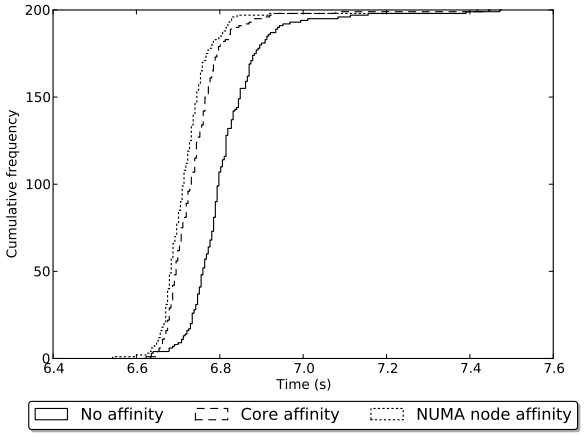
(a) 2^{26} long integers (b) 2^{27} long integers (c) 2^{28} long integers

Figure 4.4: Results for array-backed streams with disk access.

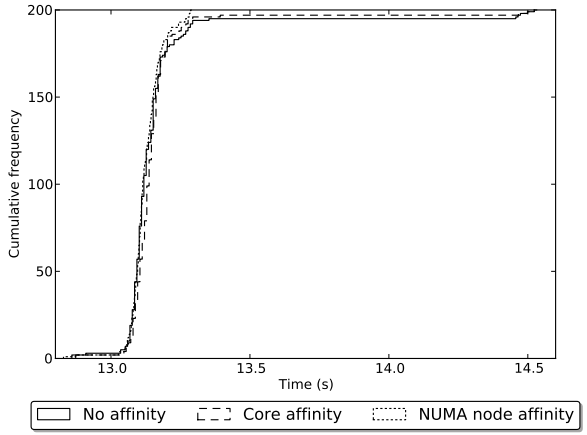


(a) 2^{26} long integers (b) 2^{27} long integers (c) 2^{28} long integers

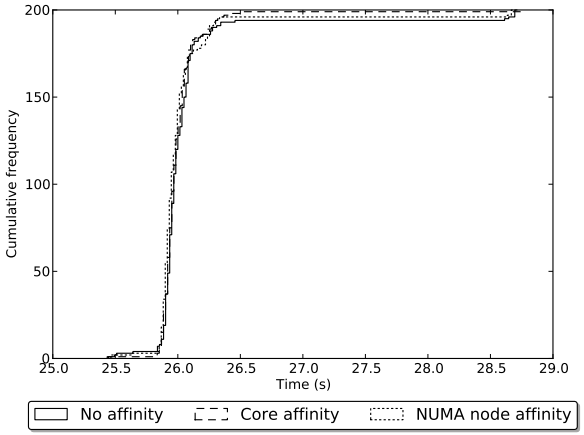
Figure 4.5: Results for Stored Collection-backed streams with disk access.



(a) 2^{26} long integers

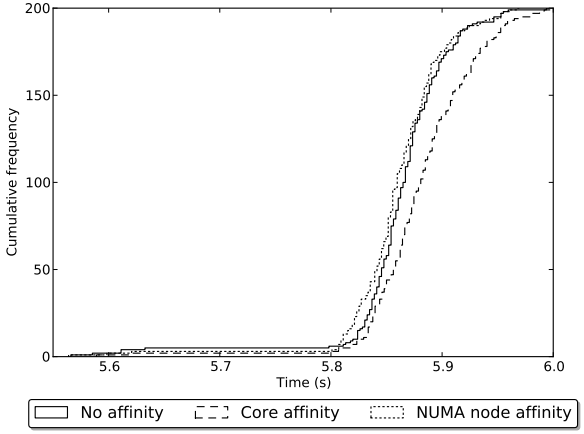


(b) 2^{27} long integers

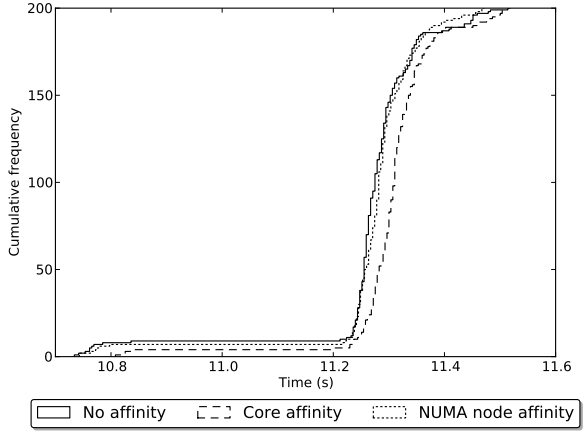


(c) 2^{28} long integers

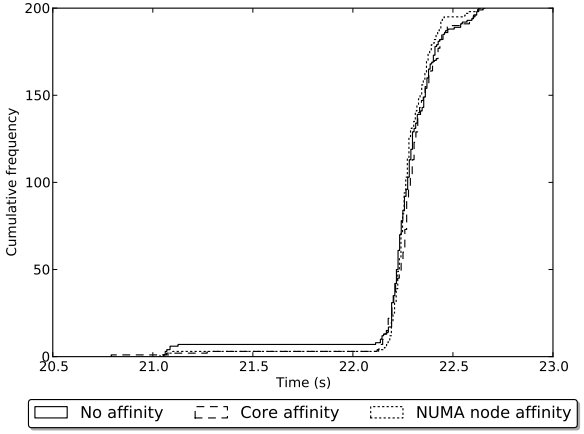
Figure 4.6: Results for array-backed streams without disk access.



(a) 2^{26} long integers



(b) 2^{27} long integers



(c) 2^{28} long integers

Figure 4.7: Results for Stored Collection-backed streams without disk access

4.4 Summary

The results of these evaluations show that besides being able to handle large datasets, Stored Collections tend to keep data more local than simple arrays (and by extension, other in-memory Java collections) within a node. The programmer also need not be concerned about optimising data locality within a node when using Stored Collections, as the system is able to do this well. Thus, Stored Collections are a suitable base to build a distributed stream source on.

This chapter has introduced and evaluated Stored Collections, which allow efficient processing of large datasets on a single node. Stored Collections avoid out-of-memory problems by reading data from disk on demand. They also improve the locality of data compared to in-memory data sources, and are able to do this without help from the programmer. This first step towards Big Data addresses the large dataset and data locality requirements for single nodes (see requirements 2 and 3 in section 3.3). The next chapter further extends the programming model to address the rest of the requirements.

Chapter 5

The Distributed Stream Framework

Chapter 3 listed several requirements for a Big Data framework based on Java 8 Streams in section 3.3, and chapter 4 proposed the concept of Stored Collections to handle large datasets on a single node in section 4.1. This chapter addresses the rest of the requirements, in particular, to support distributed cluster-based computing while maintaining backward compatibility with the existing framework. It does so by introducing the *Distributed Stream framework*, whose programming model inherits Java 8 Streams' global-view and pipeline-based model but extends it to support cluster-based computation. Figure 5.1 illustrates the high-level view of the framework. It introduces several components that are visible to the programmer:

- Stored Collections and Distributed Collections, which build on the existing Java collections framework.
- Compute nodes and groups, which utilise a communication layer to synchronise execution with other nodes.
- Distributed Streams, which build on the existing Java 8 Stream framework and the communication layer for distributed computation.

Section 5.1 describes the approach taken for the framework's design, while the details are explained in sections 5.2 to 5.4. Section 5.5 presents examples of using Distributed Streams and, finally, section 5.6 concludes the chapter.

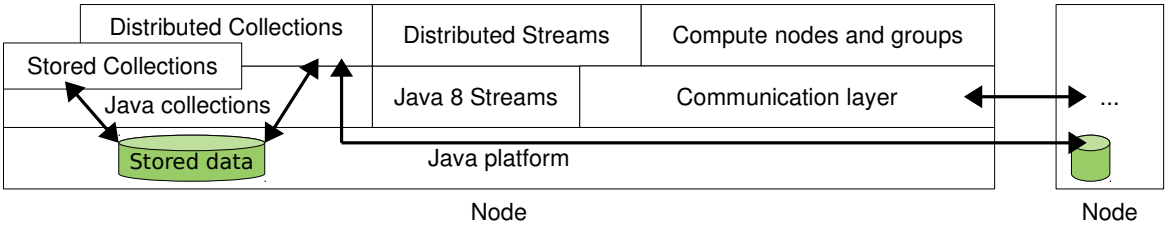


Figure 5.1: High-level view of the Distributed Stream framework. Arrows indicate flow of data between stored data and Stored/Distributed Collections as well as between the communication layers.

5.1 Approach

Since Java 8 Streams only function within a node, a distributed approach for computation is needed. As multiple machines are involved, a system of identifying and grouping them is first necessary. Section 5.2 explains how nodes in a cluster are represented in the framework.

Chapter 4 showed how Java's collections can be extended to support reading large datasets (as Stored Collections). Likewise, they can also be extended to group data that resides on separate nodes together, allowing distributed datasets to be represented. Section 5.3 describes this extension to Java's collections – *Distributed Collections*.

As with collections, a Stream can similarly be extended to a distributed environment. There are several methods to extend a pipeline-based stream framework:

- Replicating the pipeline – Identical copies of a pipeline can be run on multiple compute nodes, each processing local data from a distributed dataset.
- Distributing the pipeline – The pipeline can be broken up into sections, each running on separate compute nodes. Different data items can be processed at the same time on different nodes, and data processed by one pipeline section is transferred to the next.
- A combination of replicating and distributing the pipeline.

Pipeline replication and distribution are illustrated in figure 5.2.

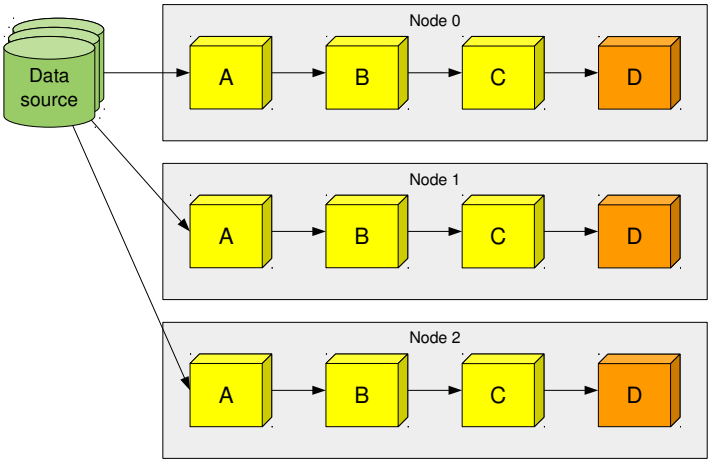
Both methods potentially speed up processing, depending on the workload. An I/O-bound workload will see a speed improvement with a replicated pipeline as as result of data locality. On the other hand, a CPU-bound workload will benefit more with a distributed pipeline, because the processing of a single pipeline is shared among multiple compute nodes.

Thus, by supporting both replicated and distributed pipelines, both types can be specified according to the workload's attributes. Section 5.4 explains the concept and design of *Distributed Streams*, which are central to the framework. It also details the backward-compatible nature of the framework, as well as additional concepts to support distributed computing.

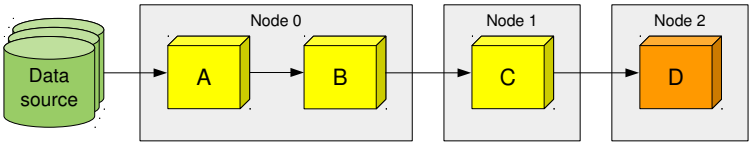
To extend the operations in Java 8 to a distributed environment, they are grouped into those that require communication across nodes (non-local) and those that do not (local). For non-local operations, new local variants are added to allow for greater expressiveness of the model and optimisation within a node (see section 5.4.1 for more about the new local operations). A general-purpose `distribute` operation is also added to allow for situations where data needs to be shuffled across the cluster or transferred to another group of compute nodes (this is detailed in section 5.4.2). This allows specifying pipelines such as that shown in figure 5.3.

5.2 Compute nodes and groups

Before distribution of data and computation can be supported, a system of identifying and grouping compute nodes is needed for decisions such as which node (or group of nodes)



(a) A replicated pipeline.



(b) A distributed pipeline.

Figure 5.2: Difference between replicating and distributing a pipeline of four operations (A, B, C and D) over three nodes. Arrows represent data flow.

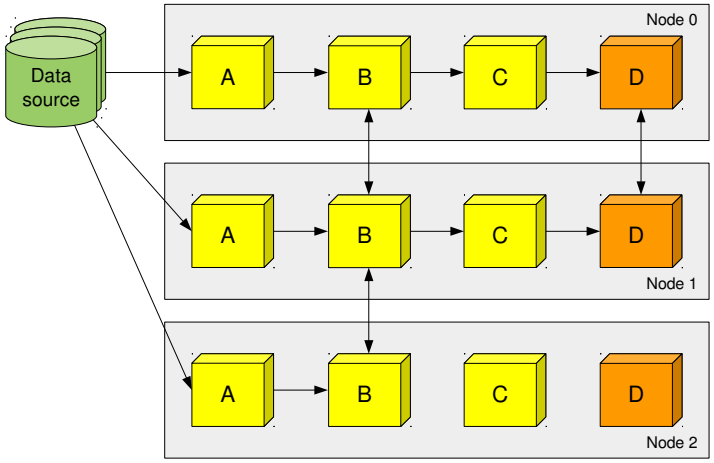


Figure 5.3: Example Distributed Stream pipeline. Operations A and C are data-parallel, while operations B and D require communication between nodes. Arrows represent data flow.

to send data to. This is achieved with the `ComputeNode` and `ComputeGroup` classes. An example usage can be found in listing 5.10.

5.2.1 The `ComputeNode` class

Each node in the cluster is represented by a `ComputeNode` object and has a unique name (the model does not prescribe a node naming convention). There are also methods to get the current node, or a specific node by its name. When the program is initialised, each compute node starts a single JVM where all computation on that node takes place.

```
package dstream;

public class ComputeNode {
  public String getName();
  public boolean isSelf();

  public static ComputeNode getSelf();
  public static ComputeNode findByName(String name);
}
```

Listing 5.1: The `ComputeNode` class.

5.2.2 The `ComputeGroup` class

A compute group is an ordered set of compute nodes and is represented by a `ComputeGroup` object.

```
package dstream;

public abstract class ComputeGroup extends ArrayList<ComputeNode> {
  public ComputeGroup();
  public ComputeGroup(Collection<ComputeNode> nodes);
  public ComputeGroup(ComputeNode node);

  public static ComputeGroup getCluster();
}
```

Listing 5.2: The `ComputeGroup` class.

The entire cluster can be retrieved by calling the `getCluster` method. Standard Java `List` methods can be called to modify a compute group's members. To prevent modifications of one compute group affecting another, the `ComputeGroup(nodes)` constructor makes a copy of given collection, and the `getCluster` method returns a new compute group for each call.

The `ComputeGroup` class is defined as abstract, in order to support different implementations of compute groups. For example, if fault tolerance is needed, an implementation can provide a subclass of `ComputeGroup` which monitors all compute nodes and handles failures gracefully. If not, a simple but fast implementation would be sufficient. The introduction of compute nodes and compute groups exposes an SPMD paradigm to the programmer.

Compute nodes and groups are explicitly used when it is necessary for data to be moved to a certain node (or subset of nodes) during computation. However, it is expected

in the common case that computations occur across the entire cluster, and this is assumed if the compute node/group is omitted in methods that accept them. A program that does not make explicit use of compute nodes and groups is more portable across clusters as it does not assume the existence of certain nodes. It is also likely to be more scalable as computation and data are distributed across all compute nodes.

5.3 Distributed Collections

This section describes extensions to Java's collections[1], called *Distributed Collections*, for use as distributed data sources.

A Distributed Collection contains data that is partitioned across a cluster. It is used as a source of a Distributed Stream, the same way as a collection is a source of a Java 8 Stream. Data items in a Distributed Collection correspond to those in the underlying dataset. The `DistributedCollection` interface itself does not specify the data format and how the data should be retrieved. Instead, this is done in subclasses to the interface. This approach is taken as there are already many existing distributed data formats in use, and it is more extensible to allow any format to be accessed through a subclass than to force one (or a few) formats on the framework.

5.3.1 Distribution of data

Section 4.1 introduced Stored Collections, which extend Java's in-memory collections to efficiently read large datasets from a local disk. However, neither Stored Collections nor Java's existing collections address the issue of data distribution across a cluster, which is an orthogonal problem.

A Java collection, being an in-memory dataset, implicitly guarantees that all its contents are in the Java heap. A Stored Collection breaks this guarantee, but still ensures that all its contents will be accessed when iterating through the dataset. However in a Distributed Collection, the dataset is partitioned across compute nodes, so iteration on a single node will only yield part of its contents. (Thus, to access the entire dataset, all participating nodes will need to iterate through the Distributed Collection.)

The `DistributedCollection` interface extends Java's `Collection` interface, enabling two main use cases. The first use case is to extend existing (non-distributed) collections, without needing to re-implement the distributed version from scratch. Conceptually, this groups data on each compute node (in the existing collections) and treats them as a single, distributed dataset. The second, more complex, use case is for datasets that are not already or not easily encapsulated in Java collections. These datasets (such as HDFS files) often have their own interfaces for data storage and retrieval, may also have built-in fault tolerance and data distribution, or may even be stored remotely. Such Distributed Collection implementations will need to take these attributes into account.

A data element that exists in a Distributed Collection is in exactly one of its compute nodes, thus the programmer does not need to handle coherency issues in the data.

[1]Since maps are also part of Java's collection framework, this section also applies to distributed maps.

However, the underlying data format can provide redundancy as long as this is abstracted away by the Distributed Collection implementation.

The Distributed Collection interface is shown in listing 5.3. This extends a Java collection as follows:

- The overridden `stream` and `parallelStream` methods now return a `DistributedStream` object instead of a `Stream` object.
- There is an associated compute group containing the nodes on which its data can be found. The `getComputeGroup` method returns this information.
- The `wrap` methods create a new Distributed Collection by grouping together normal (non-distributed) collections across nodes.

Using `wrap` is a quick way to group an existing collection on each compute node into a new Distributed Collection, which can then be used as a Distributed Stream source. In such a Distributed Stream, the pipeline on each compute node gets data from the local collection for processing. The two `wrap` methods are essentially identical in behaviour. If the compute group is specified, local collections on compute nodes in the group are used. If the compute group is omitted, the entire cluster is assumed.

```
package dstream.util;

public interface DistributedCollection<E> extends Collection<E> {
  public ComputeGroup getComputeGroup();
  @Override public DistributedStream<E> stream();
  @Override public DistributedStream<E> parallelStream();
  public static DistributedCollection<E> wrap(Collection<E> c, ComputeGroup grp);
  public static DistributedCollection<E> wrap(Collection<E> c);
}
```

Listing 5.3: The Distributed Collection interface.

5.3.2 Caching intermediate results

It is sometimes necessary to avoid recomputing data by storing intermediate results. Java 8 Streams already allow storing data into a collection by using the `collect` operation. (This terminates the stream, so a new stream consisting of the stored data needs to be created.) Some forms of the `collect` operation allow the programmer to specify the storage container, which can be a Distributed Collection. Alternatively, the `DistributedCollection.wrap` methods can be used to group the resulting collections on each compute node into a Distributed Collection.

Distributed Collections that are derived from existing Java collections will support in-memory caching with little or no implementation effort, because it is already possible to add data items to each local collection. However, support for caching on other Distributed Collections depend on the underlying data storage system. For example, HDFS files are write-once; thus, caching to a Distributed Collection backed by an existing HDFS file is prohibited.

5.4 Distributed Streams

A Distributed Stream is a sequence of data items that is spread over a subset of the cluster, with a Distributed Collection being its data source. It is used in place of a Stream when performing computations on a cluster. Pipelines that are constructed from a Distributed Stream are replicated over the respective compute nodes, enabling cluster-level data parallelism. This allows each node to perform the same pipeline operations on their local data items.

Each Distributed Stream has an associated compute group, which is initialised to that of the source Distributed Collection. In most cases this will be the entire cluster. If data needs to be relocated to other nodes, this should be done in the pipeline (using the `distribute` operation introduced in section 5.4.2).

Distributed Streams extend the functionality of Java 8 Streams, fulfilling the distribution and drop-in replacement requirements described in section 3.3.

5.4.1 A drop-in replacement for Java 8 Streams

To maintain compatibility, Distributed Streams are a drop-in replacement for Java 8 Streams. The new `DistributedStream` interface extends the existing Java 8 `Stream` interface, overriding methods that return `Stream`s with those that return `DistributedStream`s. A partial definition of the Distributed Stream interface is in listing 5.4.

For completeness, primitive-type `Stream` interfaces (`IntStream`, `LongStream` and `DoubleStream`) will have corresponding Distributed Stream interfaces (`DistributedIntStream`, `DistributedLongStream` and `DistributedDoubleStream`).

Some operations require communication between nodes to ensure correctness of results (the `count` operation for example). This is solved by overriding these operations so that they behave as expected. Thus, the `count` operation in Distributed Streams additionally sums up the partial results of each participating node and sends the total count back to the nodes. The `flatMap` operation, on the other hand, does not require inter-node communication for correct results and its implementation can thus be left unchanged.

The Java 8 Stream framework does not restrict the lambda expressions passed to pipeline operations. However, it discourages the use of side effects in such lambda expressions [65] as they may cause race conditions when accessing shared resources in parallel streams. This also applies to the Distributed Stream framework, with the additional caveat that the shared resources are not automatically distributed over the cluster.

Finally, making these operations work across a compute group introduces two further issues:

- For operations such as `collect`, a large result requires significant network communication to replicate data elements on each node. This is often undesirable or unnecessary.

- For cases where the programmer actually intends to obtain local results on each node (for example, to obtain a partial count for further computation), this is currently not possible.

```
package dstream;

public interface DistributedStream<T> extends Stream<T> {
  // Methods that override Stream methods
  @Override public DistributedStream<T> distinct();
  @Override public DistributedStream<T> filter(Predicate<? super T> predicate);
  @Override public <R> DistributedStream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper);
  @Override public DistributedDoubleStream flatMapToDouble(
    Function<? super T,? extends DoubleStream> mapper);
  @Override public DistributedIntStream flatMapToInt(
    Function<? super T,? extends IntStream> mapper);
  @Override public DistributedLongStream flatMapToLong(
    Function<? super T,? extends LongStream> mapper);
  @Override public DistributedStream<T> limit(long maxSize);
  @Override public <R> DistributedStream<R> map(
    Function<? super T,? extends R> mapper);
  @Override public DistributedDoubleStream mapToDouble(
    ToDoubleFunction<? super T> mapper);
  @Override public DistributedIntStream mapToInt(
    ToIntFunction<? super T> mapper);
  @Override public DistributedLongStream mapToLong(
    ToLongFunction<? super T> mapper);
  @Override public DistributedStream<T> parallel();
  @Override public DistributedStream<T> peek(Consumer<? super T> action);
  @Override public DistributedStream<T> sequential();
  @Override public DistributedStream<T> skip(long n);
  @Override public DistributedStream<T> sorted();
  @Override public DistributedStream<T> sorted(Comparator<? super T> comparator);
  ...
}
```

Listing 5.4: The Distributed Stream interface with methods that override those in the `Stream` interface.

To address these issues, new local variants of these operations that handle data within a node are provided (see listing 5.5 for a few of these operations). Both variants behave identically on single-node clusters, but the local operations are more efficient over multiple nodes and can thus be used to construct more efficient pipelines.

```
package dstream;

public interface DistributedStream<T> extends Stream<T>
{
  ...
  // New local operations
  public <R, A> R localCollect(Collector<? super T, A, R> collector);
  public <R> R localCollect(Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner);
  public DistributedStream<T> localDistinct();
  public DistributedStream<T> localForEach(Consumer<? super T> action);
  public DistributedStream<T> localLimit(long maxSize);
  public DistributedStream<T> localPeek(Consumer<? super T> action);
  public Optional<T> localReduce(BinaryOperator<T> accumulator);
  public T localReduce(T identity, BinaryOperator<T> accumulator);
  public <U> U localReduce(U identity,
    BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);
  public DistributedStream<T> localSkip(long n);
  public DistributedStream<T> localSorted();
  public DistributedStream<T> localSorted(Comparator<? super T> comparator);
  ...
}
```

Listing 5.5: Local operations in the Distributed Stream interface.

5.4.2 Distribution of data and computation

As mentioned in section 5.4, Distributed Streams use replicated pipelines for data parallelism by default. This handles the most common case for Big Data, where the data source is distributed over compute nodes.

A Distributed Stream sources its data from a Distributed Collection, which reads the local data items on each compute node (as described in section 5.3.1) and passes them through the pipeline on the same node. This ensures data locality, with each compute node processing data stored on its disk. However, at certain points in the pipeline, it may be required for data to be distributed and gathered over the compute nodes to do partitioning and aggregation, such as in the MapReduce shuffle stage. Java 8 Streams support gathering of data with the general-purpose terminal operations `collect` and `reduce`, but there are currently no operations for distributing data according to a given algorithm (as there is no requirement to do this in the current Java 8 Streams). Therefore, a set of operations called `distribute` which facilitate the transfer of data using inter-node communication is introduced.

The Distributed Stream model does not require that all participating compute nodes execute exactly the same operations in the evaluation of a pipeline. For example, a pipeline can read data on a given set of input nodes, filter it on a different set of compute nodes, and then store it on a further different set of output nodes. (See section 5.5.2 for a more concrete example.) The `distribute` operation can thus be viewed as moving the

evaluation of part of the pipeline from one subset of the cluster to another. (The subsets may overlap and may even be the same.)

In order to support distribution of data across the cluster, it is first necessary to provide some standard interfaces that allow the programmer to specify how the data should be partitioned. The `Partitioner` interface (shown in listing 5.6) allows user-specified partitioning of data. Similar interfaces for primitive-type partitioners (`IntPartitioner`, `LongPartitioner` and `DoublePartitioner`) for the appropriate primitive-type Distributed Streams are also defined.

```
package dstream;

@FunctionalInterface
public interface Partitioner<T> {
  public int partition(T data);
}
```

Listing 5.6: The `Partitioner` interface.

The `partition` method accepts a data element and returns an index representing a node in the compute group. For ease of use, it is not important for the programmer to know the size of the compute group, so if the index is out of range it will be wrapped around by the framework. This makes the partitioner suitable for range partitioning if the compute group size is known, as well as load balancing methods such as hash-based partitioning.

With a way to partition data, several variants of the `distribute` method that transfer data from one compute group to another can be introduced in the `DistributedStream` interface (see listing 5.7).

```
package dstream;

public interface DistributedStream<T> extends Stream<T> {
  ...
  // Methods for distribution of data
  public DistributedStream<T> distribute();
  public DistributedStream<T> distribute(Partitioner<? super T> p);
  public DistributedStream<T> distribute(ComputeGroup grp);
  public DistributedStream<T> distribute(
    ComputeGroup grp, Partitioner<? super T> p);
  public DistributedStream<T> distribute(ComputeNode node);
  ...
}
```

Listing 5.7: `distribute` methods in the `DistributedStream` interface.

Parameters change the behaviour of `distribute` as follows:

- If used without parameters, `distribute` sends data to the same compute group according to a default hash-based partitioner.
- If a partitioner is given, it is used in place of the default hash-based partitioner.
- If a compute group/node is given, data is partitioned and sent to that group/node instead. The nodes in the specified compute group do not have to be part of the

initial compute group.

A `distribute` operation returns a new Distributed Stream consisting of the same data elements which may have been moved across nodes.

5.4.3 More flexible pipelines

Distributed Streams inherit pipelines from Java 8 Streams, which are linear in nature. While adequate in most cases, this hinders the concise implementation of programs which are better expressed with non-linear pipelines. These limitations can be addressed by allowing the splitting and joining of pipelines. Hence, two new operations are introduced (see also listing 5.8) to support these actions:

- The `split` operation splits the current pipeline into a specified number of pipelines. Each resulting pipeline receives the same stream of data elements from the current pipeline.
- The `join` operation merges a number of pipelines with the current pipeline. The resulting pipeline receives data elements from all pipelines.

To preserve data locality, these operations do not transfer data across nodes. However, a subsequent `distribute` operation may be used to move the data to the desired locations.

```
package dstream;

public interface DistributedStream<T> extends Stream<T> {
    ...
    public DistributedStream<T>[] split(int numStreams);
    public DistributedStream<T> join(DistributedStream<T>... streams);
}
```

Listing 5.8: The `split` and `join` operations for creating non-linear pipelines.

5.4.4 Short-circuiting evaluation

A key property of Java 8 Streams is that they are short-circuiting, but extending this to a distributed pipeline environment impacts efficiency in the following ways:

- The pipeline has to wait for existing data items to be fully processed before deciding if more data items are needed. This may result in idle pipeline sections in certain nodes.
- Requests for data items originate from the terminal operation, and for distributed pipelines this has to be communicated upstream to different nodes which increases overhead. There is also the additional problem of choosing the node(s) from which to retrieve data.

Figure 5.4 shows an example pipeline that uses a `distribute` operation, and where operations A and (terminal) C are on different nodes. If the entire pipeline is short-circuiting, operation A will wait for a message from operation C to begin processing the

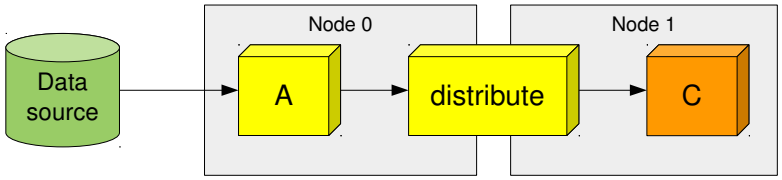


Figure 5.4: A distributed pipeline to illustrate the problem of short-circuit evaluation in a distributed environment.

next data item. The inefficiency is compounded if operation A potentially takes a long time to let a data item through the pipeline (eg. filtering). The time spent waiting in upstream operations could have been used to process data speculatively to keep those operations better utilised.

Consequentially, the solution employed is to view the `distribute` operation as terminating the current pipeline and starting another. This ensures that each section of the pipeline is on a single node and is evaluated as efficiently as possible without having to be concerned with parallelism in other pipeline sections. Therefore, this solution of restricting short-circuit evaluation to each node is simpler than enforcing it across nodes.

5.5 Examples

This section contains examples that show Distributed Stream usage.

5.5.1 Word-count

Returning to the word-count example, the drop-in replacement extensions allow code describing the pipeline to be identical in both Streams and Distributed Streams (see listings 2.3 and 5.9 for a comparison). In this case, the `lines` argument in the `wordcount` method will be a `DistributedStream`. The pipelines are also replicated on each participating compute node[2].

5.5.2 Detailed word-count

This section illustrates the use of the `distribute` operation in two examples.

The word-count example above computes the total number of words, but does not give a breakdown of the individual count for each word. To obtain a detailed word-count, the algorithm can be changed to the one in listing 5.10. (It uses Distributed Maps, which are analogous to Distributed Collections.) The steps are as follows: On each compute node, all local data is split into words and sent to a single node, where they are counted and collected into local maps of *(word, count)* pairs. These maps are then grouped as a Distributed Map using the `wrap` method. Finally, the contents of the Distributed Map are added to the `result` collection.

The algorithm above is neither efficient nor scalable, because the workload is not evenly distributed across all nodes, and the node that is handling the incoming words may run out of memory when attempting to store all the intermediate results. Listing 5.11 gives an

[2] Pipeline replication is done using the underlying communication layer, which is described in chapter 6

```
import java.util.regex.*;
import java.util.stream.*;
import dstream.*;
import dstream.util.*;

public class WordCount
{
  public static long wordcount(Stream<String> lines)
  {
    Pattern delim = Pattern.compile("\\s+");
    return lines
      .flatMap(line -> Stream.of(delim.split(line)))
      .count();
  }

  public static void main(String[] args)
  {
    String filename = args[0];
    Collection<String> c = new DistributedStringStoredCollection(filename);
    Stream<String> s = c.parallelStream();
    System.out.println(wordcount(s));
  }
}
```

Listing 5.9: Simple word-count application using Distributed Streams.

```
...
public static void wordcount(DistributedCollection<String> lines,
  DistributedCollection<String> result)
{
  Pattern delim = Pattern.compile("\\s+");
  ComputeNode rootNode = ComputeNode.findByName("node0");
  // Split data into words and send to root node
  // Process local words into a map of (word, count) pairs
  Map<String, Long> totalCount = lines
    .parallelStream()
    .flatMap(line -> Stream.of(delim.split(line)))
    .distribute(rootNode)
    .localCollect(Collectors.groupingByConcurrent(w -> w,
      Collectors.counting()));
  // Group local maps as a Distributed Map
  DistributedMap<String, Long> totalCountDist
    = DistributedMap.wrap(totalCount);
  // Add to result collection
  totalCountDist
    .entrySet()
    .stream()
    .localForEach(e -> { result.add(e.getKey() + "␣" + e.getValue()); });
}
...
```

Listing 5.10: Detailed word-count application using Distributed Streams.

improved algorithm. It first performs a local detailed word-count, and the *(word, count)* pairs are then distributed across the cluster such that pairs with the same word get sent to the same compute node (similar to the MapReduce shuffle stage). This results in a more even workload, reduced network usage from sending partially-reduced data, and a smaller chance of running out of memory.

```
...
public static void wordcount(DistributedCollection<String> lines,
  DistributedCollection<String> result)
{
  Pattern delim = Pattern.compile("\\s+");
  // Perform local detailed word-count
  Map<String, Long> localCount = lines
    .parallelStream()
    .flatMap(line -> Stream.of(delim.split(line)))
    .localCollect(Collectors.groupingByConcurrent(w -> w,
      Collectors.counting()));
  // Group local maps as a Distributed Map
  DistributedMap<String, Long> localCountDist
    = DistributedMap.wrap(localCount);
  // Shuffle (word, count) pairs and collect into local maps
  Map<String, Long> totalCount = localCountDist
    .entrySet()
    .parallelStream()
    .distribute(e -> e.getKey().hashCode())
    .localCollect(Collectors.groupingByConcurrent(e -> e.getKey(),
      Collectors.summingLong(e -> (long) e.getValue())));
  // Group local maps as a Distributed Map
  DistributedMap<String, Long> totalCountDist
    = DistributedMap.wrap(totalCount);
  // Add to result collection
  totalCountDist
    .entrySet()
    .stream()
    .localForEach(e -> { result.add(e.getKey() + "␣" + e.getValue()); });
}
...
```

Listing 5.11: Improved detailed word-count application using Distributed Streams.

5.6 Summary

This chapter has presented the components of the Distributed Stream framework, which satisfy the requirements listed in section 3.3:

- Compute nodes and compute groups for identifying and grouping nodes in a cluster. This satisfies requirement 1.
- Distributed Collections as a distributed data source and for storing intermediate results. Together with Stored Collections, this satisfies requirement 2.
- Distributed Streams with replicated pipelines for cluster-wide data processing and as a drop-in replacement for Java 8 Streams. This satisfies requirements 3 and 5.
- New Distributed Stream operations for distributing data and computation across nodes. This satisfies requirement 4.

Chapter 6

Implementation of the Distributed Stream Framework

This chapter gives an overview of how the proof-of-concept Distributed Stream framework is implemented. A high-level view of its components are shown in figure 6.1. In general, it is based on the default Java 8 Stream implementation, inheriting and extending several of its classes. This allows for future changes to the default Stream implementation without significantly affecting Distributed Streams.

Section 6.1 discusses the transport layer. Sections 6.2 and 6.3 discuss compiling and starting Distributed Stream programs. Sections 6.4 and 6.5 elaborate on compute nodes and groups. Sections 6.6 and 6.7 expand on how Distributed Collections and Distributed Streams are implemented. Finally, section 6.8 summarises.

6.1 Transport layer

Communication between nodes is handled by a transport layer. This additional layer of abstraction helps to simplify porting to other transport layers (e.g. to RMI) if necessary, without significant modification to the rest of the library.

Since there is no requirement to use any particular transport layer, a patched version of the MPJ Express [79] library is used. This is for several reasons:

- It is based on MPI, a message-passing protocol widely used in cluster-based parallel

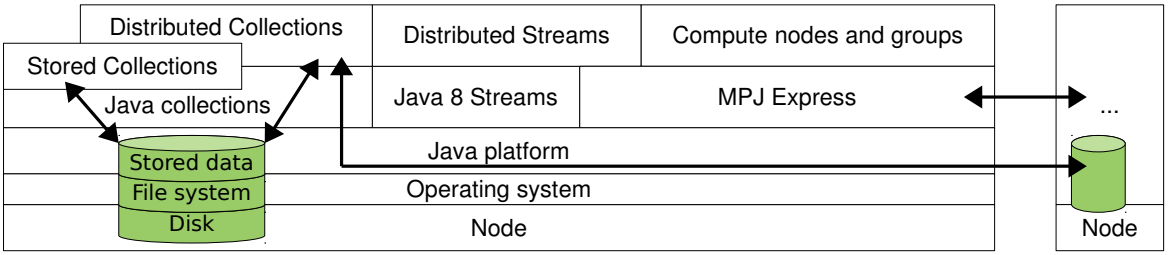


Figure 6.1: High-level view of the proof-of-concept implementation. Arrows indicate flow of data between stored data and Stored/Distributed Collections as well as between the communication layers (MPJ Express).

computing.

- It is written in Java, lightweight and has a simple API, making integration into the framework easy.
- It supports MPI in full multithreaded mode.

The prototype implementation assumes a reliable transport layer and no node failures. A more robust implementation may handle failures by configuring a timeout interval between messages sent to and received from a node, and retrying the computation, possibly on another node, if the timeout has been exceeded.

6.2 Compilation

The Distributed Stream framework and MPJ Express are provided as JAR files, thus both need to be added to the classpath.

Thus, the command to compile a simple Distributed Stream program is given in listing 6.1.

```
javac -cp .:$MPJ_HOME/lib/mpj.jar:$DSTREAM_PATH/dstream.jar HelloWorld.java
```

Listing 6.1: Command to compile a simple Distributed Stream program (in file `HelloWorld.java`). The `$MPJ_HOME` and `$DSTREAM_PATH` environment variables should specify the paths where MPJ Express and the Distributed Stream framework are installed respectively.

6.3 Program startup

Before running a program, MPJ Express needs to know the details of the cluster. This is done by creating a file (located in `$MPJ_HOME/machines`) which contains the hostnames or IP addresses of each compute node, one per line. (The machine on which this file resides does not need to be in the cluster.) The `$MPJ_HOME/bin/mpjboot` command then initialises MPJ Express on each node. In this Distributed Stream implementation, the name of each node is the string "`node`" concatenated with its rank (the first node is `node0`, the second `node1`, etc.).

A Distributed Stream program is started by executing the MPJ Express run script as shown in listing 6.2. The `-dev hybdev` flag tells MPJ Express to run across a cluster in multithreaded mode. The Distributed Stream JAR file is included in the classpath. Finally, the desired program (class name) and any program arguments are appended.

```
$MPJ_HOME/bin/mpjrun.sh -dev hybdev -cp .:$DSTREAM_PATH/dstream.jar \
  dstream.ComputeNode HelloWorld [args...]
```

Listing 6.2: Command to run a Distributed Stream program (in class file `HelloWorld.class`). The `$MPJ_HOME` and `$DSTREAM_PATH` environment variables should specify the paths where MPJ Express and the Distributed Stream framework are installed respectively.

MPJ Express first distributes the same Java bytecode to each compute node. Then, on each compute node, MPJ Express starts a JVM with the `ComputeNode.main` method as the entry point. The `ComputeNode.main` method initialises the transport layer and, once initialised, executes the actual program (in this case `HelloWorld.main`). Section 6.4 has more details on this method.

6.4 Compute nodes

As mentioned in section 6.3, the `ComputeNode.main` method is the entry point for a Distributed Stream program in this implementation. To keep transport layer-specific code separate from the `ComputeNode` class, a `Communicator` class is defined, as well as an extension for MPJ Express called `MPJCommunicator`. This communicator is then used in the `ComputeNode` class to send messages. This is illustrated in listing 6.3.

```
package dstream;
...

abstract class Communicator
{
  abstract public String[] init(String[] argv);
  abstract public void cleanup();

  abstract public int getRank();
  abstract public int getSize();

  abstract public void sendObject(Object obj, int dst, int tag);
  abstract public Object recvObject(int tag);
  // Returns source rank:
  abstract public int recvObject(Object[] obj, int tag);

  abstract public void sendInt(int n, int dst, int tag);
  abstract public int recvInt(int src, int tag);
}

class MPJCommunicator extends Communicator
{
  ... // Implementation for MPJ Express as transport layer
}

public class ComputeNode
{
  static Communicator comm;

  public static void main(String[] argv)
  {
    ...
    comm = new MPJCommunicator(); // Create this node's communicator
    ...
  }
}
```

Listing 6.3: The `Communicator` class and its uses.

The `Communicator` class defines several methods:

- `init` and `cleanup` for initialising and finalising the transport layer.
- `getRank` and `getSize` for retrieving the node's rank and the cluster size.

- `sendObject` and `recvObject` for sending and receiving messages in the form of Java objects. For MPJ Express, these objects must be serialisable (i.e. they must implement the `Serializable` interface).

- `sendInt` and `recvInt` for sending and receiving integer messages. These methods are used during initialisation of the transport layer.

Each communicator is assigned a rank (similar to that in MPI) which is used in the source and destination nodes for messages.

This separation of code is intended to confine changes to the compute node or transport layer to one class. Thus, ports to other transport layers will mainly need to implement a different communicator class and change the definition of `ComputeNode` to extend that class instead.

The `ComputeNode.main` method performs the following steps:

1. It initialises the transport layer.

2. It populates a compute group with all compute nodes in the cluster.

3. It loads the class specified on the command line and calls its `main` method.

4. Upon returning, it cleans up the transport layer.

6.5 Compute groups

Introduced in section 5.2.2, a compute group represents a source or destination of pipeline operations. It is an ordered set of compute nodes, and the `ComputeGroup` class extends an `ArrayList` of `ComputeNodes`. To maintain the constraints of an ordered set, the `ComputeGroup` class overrides the `add` and `addAll` methods to prevent duplicate compute nodes from being added to the group.

The default compute group implementation assumes no node failures and thus does not check for node connectivity or modify the set of compute nodes on its own.

6.6 Distributed Collections

The `DistributedCollection` interface overrides the default methods `stream` and `parallelStream`. It also implements new static `wrap` methods which return a `WrappedDistributedCollection` of the same data type (see listing 6.4 for more details). It keeps a reference to the local collection, which is used when generating iterators and spliterators, or retrieving the local size of the collection.

A pair of Distributed Collections (`HDFSStringCollection` and `HDFSStringCollectionWriter`) that read from and write to HDFS are implemented for the evaluating the framework in chapter 7. They are described in more detail below.

```
package dstream.util;
...

public interface DistributedCollection<E> extends Collection<E>
{
  public static <E> DistributedCollection<E> wrap(Collection<E> c,
    ComputeGroup grp)
  {
    return new WrappedDistributedCollection(c, grp);
  }

  public static <E> DistributedCollection<E> wrap(Collection<E> c)
  {
    return new WrappedDistributedCollection(c, ComputeGroup.getCluster());
  }

  ...
}

class WrappedDistributedCollection<E> extends AbstractCollection<E>
  implements DistributedCollection<E>
{
  private ComputeGroup grp;
  private Collection<E> c;

  public WrappedDistributedCollection(Collection<E> c, ComputeGroup grp)
  {
    this.c = c;
    this.grp = grp;
  }

  @Override
  public ComputeGroup getComputeGroup()
  {
    return grp;
  }

  @Override
  public Iterator<E> iterator()
  {
    return c.iterator();
  }

  @Override
  public Spliterator<E> spliterator()
  {
    return c.spliterator();
  }

  @Override
  public int size()
  {
    return c.size();
  }

}
```

Listing 6.4: Implementation of the `WrappedDistributedCollection` class and usage in `DistributedCollection.wrap` methods.

6.6.1 The `HDFSStringCollection` class

A `HDFSStringCollection` allows HDFS paths to be used as data sources for Distributed Streams. This represents a Distributed Collection of lines from all files in a specified HDFS path. Since HDFS files are already partitioned across the cluster, each compute node queries the HDFS namenode for a list of local file blocks and reads lines of text from these blocks to preserve data locality. Lines that straddle multiple blocks are automatically handled by HDFS, which sends the non-local part across the network.

6.6.2 The `HDFSStringCollectionWriter` class

HDFS files can only be written to once, so adding data to a `HDFSStringCollection` (which implies writing to an existing file) is prohibited. To allow writing of data back to a HDFS file, the `HDFSStringCollectionWriter` class is implemented. Instantiating such an object creates a new file for writing, and writes occur when the `add` method is called.

6.7 Distributed Streams

This section first details how Distributed Streams are implemented in terms of Java 8 Streams, then describes the implementation of different pipeline operations.

6.7.1 Interfacing with Java 8 Streams

The Java 8 Streams internally define several classes that implement the stream interfaces:

- `ReferencePipeline` (implementing `Stream`) for processing a stream of objects.
- `IntPipeline` (implementing `IntStream`) for processing a stream of integers.
- `LongPipeline` (implementing `LongStream`) for processing a stream of long integers.
- `DoublePipeline` (implementing `DoubleStream`) for processing a stream of double precision floats.

These classes are returned by intermediate operations and other stream-constructing methods (e.g. `Collection.stream`) so that additional operations can be appended to the pipeline.

Analogously, the prototype implementation also defines these classes which implement the respective Distributed Stream interfaces. In these classes, a reference of the local pipeline is kept. This may be used to implement the various operations on the replicated pipeline, depending on the type of operation. The remainder of this section gives more details on how this is achieved.

6.7.2 Local operations

Introduced in section 5.4.1, local operations process data within each node, allowing for partial results to be obtained and optimisations to be performed.

The local operations introduced in Distributed Streams simply call the corresponding operations on each local pipeline (i.e. a `localCount` operation calls the local pipeline's

`count` operation). Terminal operations return a result which may potentially be different on each compute node. Intermediate operations return one of the pipeline objects described in section 6.7.1, allowing more operations to be appended to the pipeline.

6.7.3 Overridden operations

Several Distributed Stream operations are reimplemented to satisfy the drop-in replacement requirement. This section gives examples of several of these operations.

The `reduce` operation

There are a number of `reduce` operations for each stream type, but all have implementations similar to the following:

1. A reduction is performed on local data elements with the `localReduce` operation.
2. The local result is sent to the first node in the compute group.
3. The first node receives and accumulates all local results, and sends the final result to the other nodes.
4. The other nodes receive the final result. All nodes return the same value.

The `allMatch` operation

The `allMatch` operation is expressed as a reduction with the result being the logical-AND of local `allMatch` operations on each participating node.

The `count` operation

The `count` operation is expressed as a reduction with the result being the sum of local `count` operations on each participating node.

The `distinct` operation

The `distinct` operation is expressed in terms of the `distribute` operation followed by the `localDistinct` operation. The `distribute` operation sends elements with the same hash value (which includes all identical elements) to the same node. The `localDistinct` operation then removes duplicate elements within each node.

The `forEach` operation

To be a drop-in replacement, each participating node needs to perform the specified action on every element in the Distributed Stream. Thus, each node broadcasts its local data elements to other nodes, thereby ensuring that each local stream contains elements from all nodes. However, this is unlikely to be efficient. The `localForEach` operation avoids broadcasting elements and is intended for programmers to optimise their implementations. (The `localForEach` operation is the local version of `forEach`, thus it only performs its operations on data elements that are on the node.)

The `sorted` operation

The `sorted` operation first performs a bucket sort: it samples the data to determine the range of values each participating node will accept, then distributes the data accordingly. It then locally performs an external merge sort on the data received by each node. This ensures that data which is too large to fit in memory can still be sorted (the Java 8 Stream implementation suffers from this issue as it uses an in-memory collection to facilitate the sort).

6.7.4 The `distribute` operation

As mentioned in section 5.4.4, the `distribute` operation terminates the existing pipeline and starts a new one. The following algorithm describes the core of all `distribute` operation variants.

The existing pipeline is terminated with a `localForEach` operation, which sends each element to the appropriate destination node. After all elements are sent, an end-of-data marker is sent to all participating nodes. This marker is represented by a null object in the implementation (null objects thus cannot be used as data items).

Concurrently, a new pipeline is created and converts incoming messages into data elements. It determines that no more data is available when it has received end-of-data markers from all participating nodes.

Since a stream computation blocks until the terminal operator completes, one extra thread is created to keep the two pipelines executing concurrently.

Default partitioner

The default hash-based partitioner computes the hash by calling the `Object.hashCode` method. This is generally sufficient for use in load-balancing data flow across the cluster.

6.8 Summary

This chapter has given an overview of the prototype Distributed Stream implementation. It confines the transport layer-dependent part to subclasses of the `Communicator` class, leaving the rest of the implementation transport layer-independent.

The next chapter evaluates the framework's programming model and performance.

Chapter 7

Evaluation

Chapters 5 and 6 have described the concept and implementation of Distributed Streams.

This chapter evaluates the following aspects of Distributed Streams:

- The types of parallelism the programming model supports (in section 7.1).
- Whether the programming model allows efficient description of Big Data programs, both for MapReduce-style and streaming processing. Comparisons are made with the programming models of MapReduce/Hadoop (in section 7.2), Spark and Storm (both in section 7.3).
- The performance of the prototype implementation (described in chapter 6), specifically the execution times and network usage. Comparisons are made with the performance of Hadoop and Spark (in sections 7.4 to 7.7).

A summary of the evaluation is presented in section 7.8.

7.1 Parallelism in Distributed Streams

Section 2.2 described three types of parallelism in programs. This section evaluates the Distributed Stream framework on the parallelism it supports.

7.1.1 Data parallelism

The most important type of parallelism in Big Data processing is data parallelism. While Java 8 Streams already supports this on a single JVM, Distributed Streams extends this support to a cluster. As with Java 8, data parallelism within a node is enabled by calling the `parallelStream` method on a collection to start a pipeline, or by using the `parallel` operation on an existing pipeline. On each compute node, a spliterator partitions the local data from the data source and submits each partition to the common fork-join pool for processing through the pipeline. At the cluster level, data parallelism is always enabled – data in a Distributed Collection is read across the cluster in parallel.

Listing 7.1 shows a pipeline that filters its input into another dataset, with figure 7.1 showing the corresponding data parallelism. The Distributed Collection is read in parallel on all compute nodes, filtered for non-empty lines and collected. As there is no need for

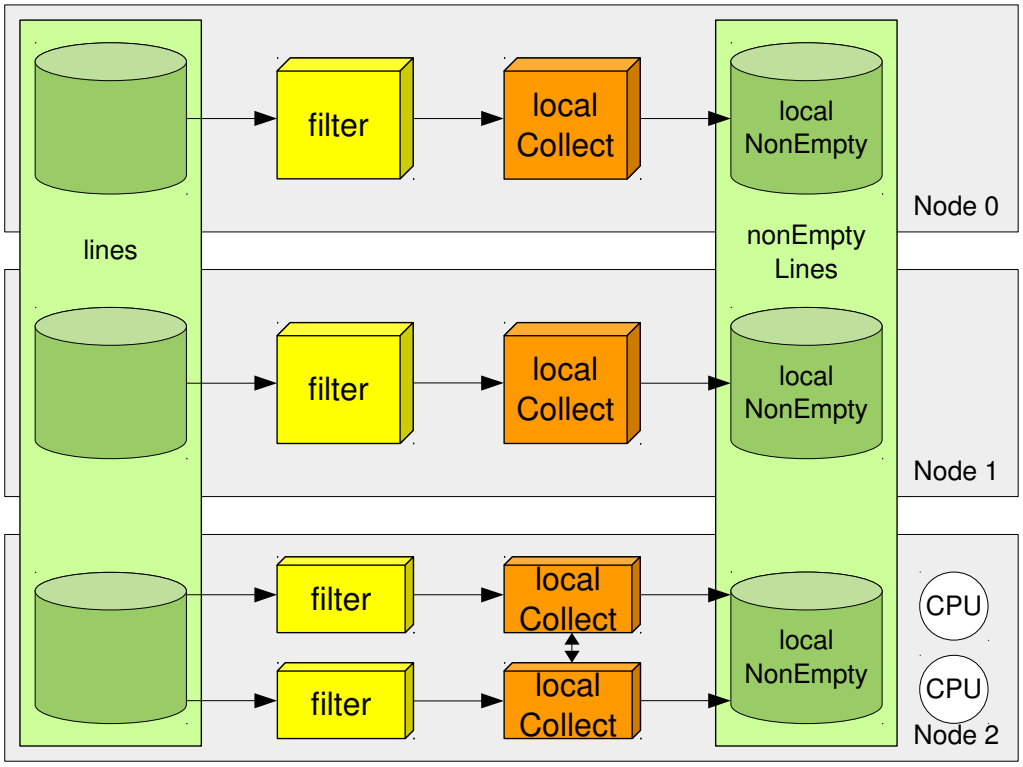


Figure 7.1: Diagram showing data parallelism in listing 7.1. Additionally, node 2 shows node-level data parallelism that is present in a multicore system.

any communication in the `filter` and `localCollect` operations, data items remain in their respective nodes.

```
DistributedCollection<String> nonEmptyLines(DistributedCollection<String> lines)
{
  Collection localNonEmpty = lines
    .parallelStream()
    .filter(s -> s.length > 0)
    .localCollect(Collectors.toList());
  return DistributedCollection.wrap(localNonEmpty);
}
```

Listing 7.1: Example pipeline that exhibits data parallelism.

7.1.2 Task parallelism

When a thread executes a Distributed Stream pipeline, it does not return control until execution is complete on all compute nodes (this is similar in behaviour to Java 8 Streams). Thus, to process multiple pipelines simultaneously, multiple threads are needed, each executing a single pipeline. Listing 7.2 demonstrates how two simple pipelines can be executed in parallel using separate threads for each pipeline, and figure 7.2 illustrates the task parallelism. The main thread creates another thread which counts the number of items in a stream, while it sums up the numbers in another stream.

For a parallel stream (created with the `parallelStream` method), the thread that initiates execution shares the workload with the fork-join thread pool on each node. For a

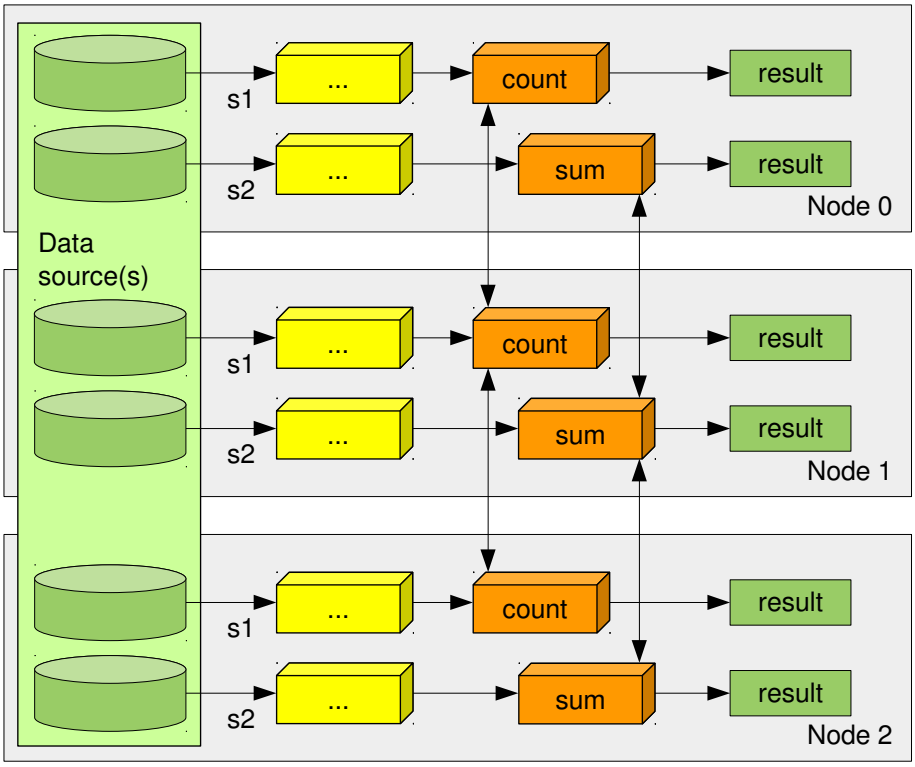


Figure 7.2: Diagram showing task parallelism in listing 7.2.

sequential stream (created with the `stream` method), the thread that initiates execution handles the entire workload on each node.

```
void main(String[] args) throws InterruptedException
{
  Thread th = new Thread(() ->
  {
    DistributedIntStream s1 = ...;
    long count = s1.count();
    ...
  });
  th.start();
  DistributedIntStream s2 = ...;
  long sum = s2.sum();
  ...
  th.join();
}
```

Listing 7.2: Executing two pipelines in parallel.

7.1.3 Stream parallelism

Due to the introduction of pipeline sections, a limited form of stream parallelism is a side-effect in Distributed Streams. Stream parallelism did not exist in Java 8 Streams because each data item was operated on by the same thread throughout the pipeline. However, recall that in section 5.4.4, the difficulties that the `distribute` operation presented were resolved by splitting the pipeline into sections. Within a pipeline section, each data item is operated on by the same thread (as in a Java 8 Stream). As pipeline sections run

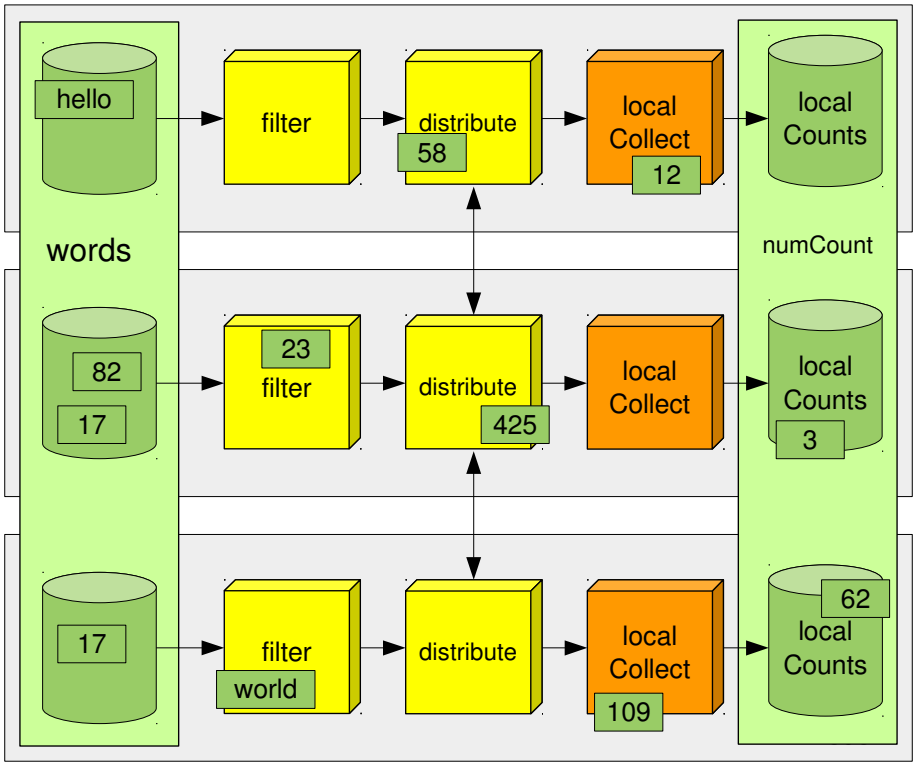


Figure 7.3: Diagram showing stream parallelism in listing 7.3. Note that the data items are processed in parallel in each pipeline section.

concurrently, each section will be operating on different data items at any time. Thus, pipelines containing the `distribute` operation exhibit stream parallelism.

Listing 7.3 shows a pipeline that counts the occurrences of numeric strings (e.g. `"35"`) from a stream of strings, with figure 7.3 illustrating the stream parallelism. The upstream pipeline section (until the `distribute` operation) filters away non-numeric strings, while the downstream section collects the remaining strings and stores them as *(number, count)* key-value pairs. (The `distribute` operation ensures that identical strings/numbers are sent to the same node.) The concurrent execution of these sections means that a thread can operate on the next item after passing it on to the `distribute` operation, and does not have to wait until the same item is collected at the terminal operation.

```
DistributedMap<String, Long> numCount()
{
  DistributedCollection<String> words = ...;
  Map<String, Long> localCounts = words
    .parallelStream()
    .filter(w -> w.matches("-?[0-9]+")) // Remove non-numeric strings
    .distribute()
    .localCollect(Collectors.groupingByConcurrent(w -> w, Collectors.counting()));
  return DistributedMap.wrap(counts);
}
```

Listing 7.3: A number-count pipeline.

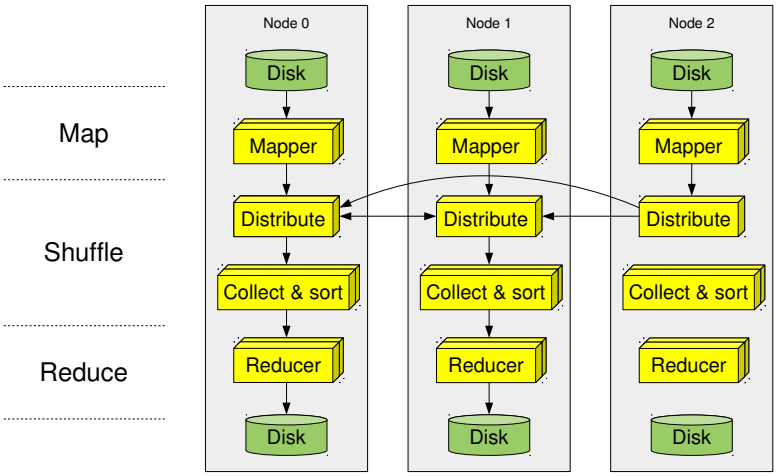


Figure 7.4: Expressing a MapReduce computation in terms of Distributed Streams. Arrows indicate the flow of data. The mapper, reducer, local collect and local sort may span multiple stream operations.

7.2 Mapping MapReduce to Distributed Streams

In section 2.5, two dominant programming models for Big Data systems were introduced: MapReduce (Hadoop) and streaming (Spark/Storm). In order to demonstrate the efficacy of the Distributed Stream programming model, comparisons between the models and Distributed Streams were carried out. This section considers how the MapReduce programming model can be expressed in terms of the Distributed Stream model.

With Java 8 Streams, performing MapReduce computations across a cluster was not possible as the framework operated within a single node. Distributed Streams solve this problem by defining operations to transfer data across nodes. In MapReduce, the shuffle stage is where data transfer is needed. This stage can be broken down into sub-stages and implemented with Distributed Streams as follows:

1. The `distribute` operation transfers data (in key-value form) between compute nodes such that those with the same key are sent to the same node.

2. The `localCollect` operation on each node accumulates incoming data into a collection of key-*value-list* pairs. The value lists are optionally sorted. Since `localCollect` is a terminal operation, a new stream consisting of elements in the collection is created and passed to the reduce stage.

Figure 7.4 shows in general how a MapReduce computation can be represented with pipeline operations. Special cases such as summing, counting and collecting are built into Distributed Streams. Hence the resulting code is more concise if such operations are used, than if programmers implement their own versions through the `reduce` operation.

7.2.1 Illustrative example

This example returns to the detailed word-count introduced in section 5.5.2. In that section, two algorithms were provided: a naive and an improved one. Listings 7.4 and 7.5 contain the programs with comments showing the equivalent MapReduce stages.

The `DistributedStringStoredCollection` referenced in both listings is a distributed version of `StringStoredCollection` mentioned in section 4.2.

Note that the shuffle stage also needs to be implemented in the Distributed Stream program. In general, the beginning and end of Distributed Stream pipelines do not necessarily correspond with MapReduce stages. In listing 7.4, the map and shuffle stages were implemented with a single pipeline, whereas in listing 7.5, all stages were implemented with separate pipelines.

In listing 7.4, the map stage consists of the `flatMap` operation which splits each line into words. The shuffle stage consists of the `distribute` operation to send all all words to the first compute node, followed by a `localCollect` operation to group words into *(word, count)* key-value pairs. Finally, the reduce stage is just a `forEach` operation to write all key-value pairs to standard output on each compute node.

In listing 7.5, the map stage consists of the `flatMap` operation as above, followed by a `localCollect` operation to group local words into a map of *(word, count)* key-value pairs. This reduces the amount of network traffic during the shuffle stage. The local maps are then wrapped into a Distributed Map (`localCountDist`), which would contain all key-value pairs in memory (this allows its entries to be used as a source of a Distributed Stream). In the shuffle stage, the key-value pairs are distributed across the cluster by key (word), collected again into *(word, count)* pairs and the resulting maps wrapped into another Distributed Map (`totalCountDist`). Finally, the reduce stage is a `localForEach` operation to write all local key-value pairs to standard output.

7.3 Mapping in-memory streaming to Distributed Streams

The previous section showed how MapReduce can be expressed in terms of Distributed Streams. This section focuses on Spark and Storm, these being the two main proponents of the streaming model.

7.3.1 Spark and comparisons with Distributed Streams

Since both Spark and Distributed Streams are stream-based programming models, there are some similarities between the models. For example, Spark pipelines are also lazily evaluated, and their Java syntax resembles that of Java 8 Streams. Also, Spark transformations are analogous to intermediate operations and Spark actions are equivalent to terminal operations.

Table 7.1 lists a number of Spark transformations/actions together with the equivalent Distributed Stream operations to demonstate the similarities between them.

There are also significant differences between the models. Spark was designed specifically for Big Data applications, whereas Distributed Streams have to maintain compatibility with Java 8 Streams. RDDs can be reused, unlike Java 8 Streams, and Spark allows caching of data in memory for frequently-used RDDs to avoid recomputing data. Also, Java 8 Streams are conceptually separate from Java collections (which can be the source of a stream), but Spark RDDs do not have such a distinction. A Spark pipeline consists of a number of RDDs chained together with transformations.Thus, depending on its position

```
import java.util.regex.*;
import java.util.stream.*;
import dstream.*;
import dstream.util.*;

public class WordCountDetailed {
  public static void wordcount(DistributedCollection<String> lines) {
    Pattern pat = Pattern.compile("\\s+");
    ComputeNode rootNode = ComputeNode.findByName("node0");
    Map<String, Long> words = lines
      .parallelStream()
    // ===== Map stage =====
      .flatMap(line -> Stream.of(pat.split(line)))
    // ===== Shuffle stage =====
      // Send to first node in compute group
      .distribute(node0)
      .localCollect(Collectors.groupingByConcurrent(
        w -> w, TreeMap::new, Collectors.counting()));
    Set<Map.Entry<String, Long>> entries = words.entrySet();
    entries
      .stream() // Generates a normal, sequential stream
    // ===== Reduce stage =====
      .forEach(e -> {
        System.out.println(e.getKey() + "\t" + e.getValue());
      });
  }

  public static void main(String[] args) {
    String filename = args[0];
    wordcount(new DistributedStringStoredCollection(filename));
  }
}
```

Listing 7.4: Naive detailed word-count application using Distributed Streams, with data funnelled to a single compute node in the shuffle stage.

```
import java.util.regex.*;
import java.util.stream.*;
import dstream.*;
import dstream.util.*;

public class WordCountDetailed {
  public static void wordcount(DistributedCollection<String> lines)
  {
    Pattern delim = Pattern.compile("\\s+");
    // Perform local detailed word-count
    Map<String, Long> localCount = lines
      .parallelStream()
    // ===== Map stage =====
      .flatMap(line -> Stream.of(delim.split(line)))
      .localCollect(Collectors.groupingByConcurrent(w -> w, Collectors.counting()));
    // Group local maps as a Distributed Map
    DistributedMap<String, Long> localCountDist = DistributedMap.wrap(localCount);
    // Shuffle (word, count) pairs and collect into local maps
    Map<String, Long> totalCount = localCountDist
      .entrySet()
      .parallelStream()
    // ===== Shuffle stage =====
      .distribute(e -> e.getKey().hashCode())
      .localCollect(Collectors.groupingByConcurrent(e -> e.getKey(),
        Collectors.summingLong(e -> (long) e.getValue())));
    // Group local maps as a Distributed Map
    DistributedMap<String, Long> totalCountDist = DistributedMap.wrap(totalCount);
    // Add to result collection
    totalCountDist
      .entrySet()
      .stream()
    // ===== Reduce stage =====
      .localForEach(e -> { System.out.println(e.getKey() + "␣" + e.getValue()); });
  }

  public static void main(String[] args) {
    String filename = args[0];
    wordcount(new DistributedStringStoredCollection(filename));
  }
}
```

Listing 7.5: Improved detailed word-count application using Distributed Streams.

Spark	Distributed Stream	Description

`.countByKey()`	`.collect(`
	`  Collectors.groupingBy(e ->`
	`  e.getKey(),`
	`  Collectors.counting()))`
		Action: Counts the occurrence of each key and returns a map of key-count pairs.

`.filter(p)`	`.filter(p)`	Transformation: Removes elements in the Distributed Stream that do not satisfy the predicate.
`.foreach(f)`	`.forEach(f)`	Action: Executes a function over each element.

`.groupByKey()`	`.collect(`
	`  Collectors.groupingBy(e ->`
	`  e.getKey()))`
	`.entrySet()`
	`.parallelStream()`
		Transformation: Groups key-value pairs into key-*value-list* pairs.

`.map(f)`	`.map(f)`	Transformation: Replaces each element with those from the mapping function.
`.reduce(f)`	`.reduce(f)`	Action: Reduces the elements to a single value using a binary function.
`.take(n)`	`.limit(n)` `.collect(Collectors.toList())`	Action: Returns the first `n` elements in a `List`.

Table 7.1: Comparison of Spark transformations/actions and Distributed Stream operations.

in the pipeline, an RDD may contain data from HDFS blocks, or transformed data cached in memory, or even information on how the data should be processed (to support lazy evaluation). However, a Distributed Stream pipeline section only manages data flow and is not concerned with data storage, thus each data item is always in memory as it is moved through the pipeline (or pipeline section).

Since Spark is a high-level framework, it does not have operations for distributing data across nodes with a programmer-supplied partitioner. Instead, this shuffling of data is done internally by several of the transformations and actions when required.

7.3.2 Distributed Stream implementation for Spark

Due to the similarities in programming models, parts of the Spark and Distributed Stream pipelines can be almost identical. However, programmers using Distributed Streams will need to be aware of the following:

- A feature of Spark is that it can reuse computed values in an RDD on a best-effort basis with the `cache` method. However, there is no equivalent feature in Distributed Streams. This is due to inheriting the Java 8 Stream model, which allows only linear pipelines and keeps data that is processed through the pipeline in memory, thus eliminating the need for caching. To manually store and use intermediate results, the programmer can collect data into a Distributed Collection (using `localCollect`) and start a new Distributed Stream from that collection. This is illustrated in figure 7.5.

- Since Spark has the concept of a driver program which defines the pipeline and

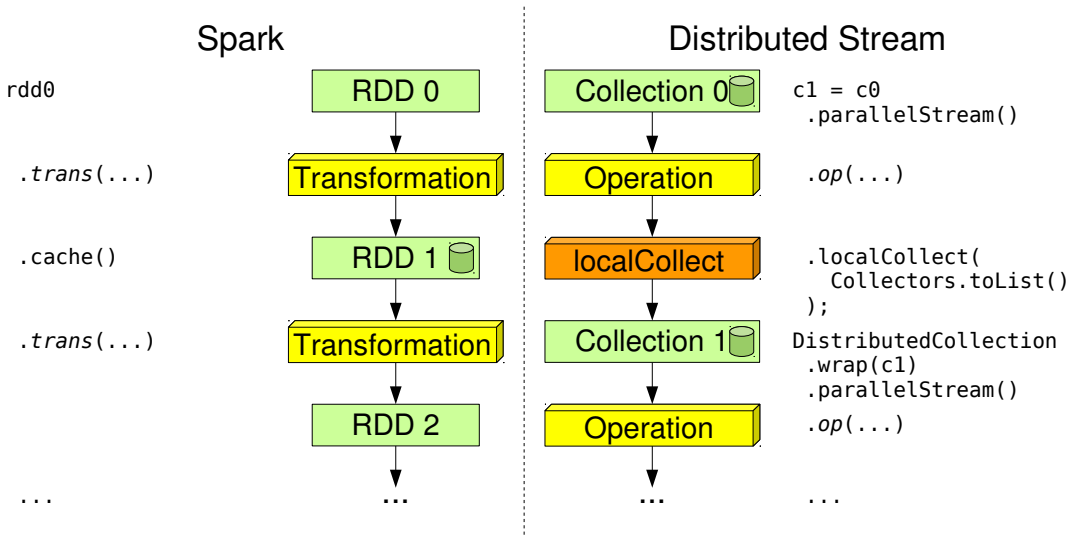


Figure 7.5: Caching in Spark and Distributed Streams. A cylinder indicates that the dataset is stored.

submits work to the master node, actions such as `collect` send data to the driver instead of to all processes on participating nodes. To achieve a similar effect of sending all data to one compute node, the `distribute(`*`node`*`)` operation can be used.

- Like Distributed Stream operations, a number of Spark transformations and actions require significant inter-node communication when dealing with datasets spanning multiple nodes and should be avoided if there are still many elements in the dataset.
- Distributed Collections are not guaranteed to be redundant. If the underlying data source has no redundancy, Distributed Collection implementations will need to handle it. This is rare, as common distributed file systems such as HDFS and Lustre [63] can be configured for redundancy, or have it enabled by default.

7.3.3 Spark example

To demonstrate the similarities and differences between Spark and Distributed Streams, reference is again made to the detailed word-count example. The corresponding Spark code (using lambda expressions) is shown in listing 7.6.

After replacing each line of text with the individual words (using `flatMap`), each word is converted to a *(word, 1)* pair. The `reduceByKey` transformation reduces values *(word, M)* and *(word, N)* to *(word, M + N)*. The `collect` action ends the pipeline, saving the remaining pairs into a `List`. Finally, the list contents are printed to standard output.

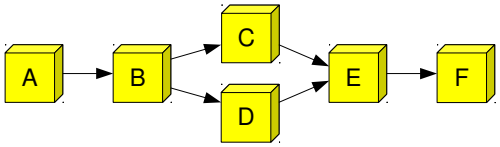
Note that Spark has built-in definitions for tuples and has transformations that handle tuples. This allows the pipeline to be more concise.

7.3.4 Storm and comparisons with Distributed Streams

Storm is also a stream-based programming model, but it performs eager evaluation, as opposed to lazy evaluation in Distributed Streams. Another major departure from the

```
static void wordcount(JavaRDD<String> lines) {
  Pattern pat = Pattern.compile("\\s+");
  List<Tuple2<String, Integer>> result = lines
    .flatMap(line -> Arrays.asList(pat.split(line))
    // Convert to (word, 1) pair
    .mapToPair(s -> new Tuple2<String, Integer>(s, 1))
    // Add pairs with same words together
    .reduceByKey((i1, i2) -> i1 + i2)
    // Save as list of (word, count) pairs
    .collect();
  // Print each (word, count) pair
  for (Tuple2<?,?> t : result)
    System.out.println(t._1() + "\t" + t._2());
}
```

Listing 7.6: Detailed word-count application using Spark.




```
DistributedStream initial = ...;
DistributedStream[] splits = initial
  .op(...) // A
  .op(...) // B
  .split(2); // Split into 2 streams
DistributedStream afterC = splits[0].op(...); // C
DistributedStream afterD = splits[1].op(...); // D
afterC
  .join(afterD) // Join streams
  .op(...) // E
  .op(...); // F
```

Figure 7.6: Example non-linear Storm topology and the Distributed Stream pseudocode that implements it.

other programming models is that Storm topologies run indefinitely, thus stopping the computation requires external action.

7.3.5 Distributed Stream implementation for Storm

In a Storm topology, bolts receive data streams consisting of tuples (from spouts and other bolts), perform computation on them, and may output streams to other bolts. A linear topology maps easily to a Distributed Stream pipeline, which is implicitly linear.

A non-linear topology can be broken up into linear sections, splits and joins. As above, a linear section can be implemented with a single Distributed Stream pipeline. A split can be implemented with the `split` operation, which sends data to multiple pipelines. A join can be implemented by the `join` operation, which merges data from multiple pipelines. Figure 7.6 illustrates an example non-linear Storm topology and the corresponding mapping to Distributed Streams.

7.3.6 Storm example

For Storm, the pipeline is split into two bolts shown in listing 7.7. The bolts are incorporated into a topology with the code in listing 7.8.

```
public class Splitter implements IRichBolt {
  // Private variables are initialised in the prepare() method
  private OutputCollector collector;

  @Override public vold execute(Tuple line) {
    Pattern pat = Pattern.compile("\\s+");
    for (String word: pat.split(line.getString(0)))
      collector.emit(new Values(word)); // Send each word to next bolt
    collector.ack(line);
  }

  // ...
}

public class Counter implements IRichBolt {
  // Private variables are initialised in the prepare() method
  private OutputCollector collector;
  private TreeMap<String, Integer> wordcount;

  @Override public vold execute(Tuple t) {
    String word = t.getString(0);
    if (wordcount.containsKey(word))
      wordcount.put(word, wordcount.get(word) + 1); // Update occurrence
    else
      wordcount.put(word, 1); // Add new word
    collector.ack(t);
  }

  // ...
}
```

Listing 7.7: Detailed word-count application using Storm.

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("input", ...);
builder.setBolt("split", new Splitter()).shuffleGrouping("input");
builder.setBolt("count", new Counter()).shuffleGrouping("split");
// ...
```

Listing 7.8: Storm topology for detailed word-count application.

When run, the `Splitter` accepts lines of text and outputs individual words. Concurrently, the `Counter` adds each word from the `Splitter` into a `TreeMap` or updates its occurrence if already present. Since Spark topologies run indefinitely, an external signal, special marker or timeout is needed to indicate the end of input and that the `TreeMap` is ready to be output.

7.4 Evaluating the performance of Distributed Streams

Direct comparisons of end-to-end execution times for Distributed Streams, Spark and Hadoop are often not appropriate because each system performs different styles of computation, at different times, and with different data distribution, replication, and fault

tolerance guarantees. Therefore, this evaluation attempts to remove initial distribution from the comparison and focus solely on the computation by pre-distributing the input data. This is how Hadoop (and Spark on HDFS) works normally, but it is not required by Storm or the Distributed Streams approach.

7.4.1 Benchmarking suite, clusters and workloads

BigDataBench [87] is a benchmarking suite containing a wide range of Big Data workloads that run on datasets randomly generated from real-world data. Unlike other benchmarking suites, it comes with a variety of real-world datasets, with workloads spanning multiple application domains such as search engines, e-commerce and social networking (its emphasis is on Internet services, but there are also workloads for DNA sequencing and ongoing progress on adding image, video and raytracing workloads). It also has multiple framework implementations for many of the workloads.

The evaluation was based on a subset of BigDataBench workloads with Hadoop and Spark implementations, and that span many of the domains covered. Being a work-in-progress, there is a possibility of testing more workloads and frameworks as they are added to BigDataBench.

Tests were carried out on two clusters with varying numbers of nodes:

1. A cluster with a master node and 6 compute nodes, with a total of 12 compute cores. This was a private cluster with exclusive access given, allowing for more accurate measurements.

2. A national, shared cluster[1] with 332 nodes, of which:

   (a) A master node and 10 compute nodes were used, with a total of 160 compute cores.

   (b) A master node and 20 compute nodes were used, with a total of 320 compute cores.

   Using two cluster sizes allows evaluating the framework's scalability as more nodes are added. However, the shared nature of this cluster made measurements less accurate (fewer repetitions were carried out due to resource constraints, and there was the potential of interference from other users).

Table 7.2 lists the workloads that were evaluated, along with their attributes and the input sizes used. For each workload, a Distributed Stream application was implemented and compared against the provided Hadoop and Spark implementations.

In the BigDataBench suite, the Hadoop and Spark workloads read their data from HDFS. Thus, to use the same data source for all tests, `HDFSStringCollection`s and `HDFS-StringCollection-Writer`s were implemented to allow Distributed Stream workloads to also access HDFS data (see section 6.6.1 and 6.6.2 for details on their implementation).

[1] N8 High Performance Computing cluster (URL: `http://n8hpc.org.uk`).

Workload	Attributes	Inputs (cluster 1)	Inputs (cluster 2)
Grep	Disk-intensive	2GB – 32GB	2GB – 32GB
Sort	Disk-intensive, communication-intensive	2GB – 32GB	2GB – 32GB
Word-count (detailed)	Disk-intensive	2GB – 32GB	2GB – 32GB
Bayes (naive)	Disk-intensive	1GB – 16GB	1GB – 32GB
Connected components	Graph, communication-intensive	2^{10} – 2^{18} vertices	2^{10} – 2^{20} vertices
PageRank	Graph, communication-intensive	2^{10} – 2^{18} vertices	2^{10} – 2^{20} vertices

Table 7.2: Workloads evaluated, attributes and input sizes tested.

7.5 Experimental setup

Cluster 1 consisted of one master and six compute nodes:

- Intel Core 2 Duo E6600 with 2GB RAM (master)
- Intel Core 2 Duo E6400 with 2GB RAM
- Intel Core 2 Duo E6400 with 4GB RAM
- Intel Core 2 Duo E8400 with 8GB RAM
- 3 × Intel Core 2 Duo E8400 with 4GB RAM

Each node had a single 7200RPM hard disk attached. The master node ran Ubuntu 12.04 Linux while the compute nodes ran Debian 7 Linux. All nodes were connected with gigabit Ethernet via a switch.

Cluster 2a consisted of one master and 10 compute nodes, with two Intel E5-2670 CPUs per node (for a total of 16 cores per node). Each node had 64 GB of RAM, an attached 7200RPM hard disk, and ran CentOS 6.7 Linux. All nodes were connected with InfiniBand. Cluster 2b consisted of one master and 20 compute nodes with the same hardware as cluster 2a.

For all clusters, Hadoop version 1.2.1 and Spark version 1.3.0 were used. Java 8u5 was used on cluster 1, while Java 8u45 was used on clusters 2a and 2b.

To minimise any differences in input data access, all tests read their input data from HDFS (as this was the default for the Hadoop and Spark tests) with data replication disabled. Input data was copied to HDFS before any measurements began, and the OS buffers were cleared before each test. The master node acted as the HDFS namenode as well as the jobtracker, while the compute nodes were each HDFS datanodes and tasktrackers.

7.6 Tests and workloads

The following metrics were obtained:

- Execution time – the elapsed time of the computation, excluding the time taken to copy data to HDFS. This was taken from output of the `time` command prepended to the startup command on the driver node.

- Network usage – the total volume of data transferred over the network during the computation, again excluding the initial copying of data to HDFS. This was taken from output of the `ifconfig` commands which ran before and after each run on each node, the sum of all (transferred) network traffic giving the desired metric.

A number of runs were carried out for each input size, and the average values and standard deviations were obtained:

- For cluster 1, workloads were run 15 times per input size.
- For clusters 2a and 2b, workloads were run 5 times per input size (due to resource constraints).

The rest of this section gives details of all workloads. Full code for all workloads can be found in appendix B.

7.6.1 Grep

The *grep* workload behaves similarly to the Unix `grep` tool: it outputs the lines of an input file for which the given regular expression can be found. This is implemented with Distributed Streams as shown in listing 7.9.

```
void grep(DistributedCollection<String> lines, String re)
{
  // Initialise regular expression pattern
  Pattern match = Pattern.compile(re);
  lines
    .parallelStream()
    // Retain lines that match regular expression
    .filter(line -> match.matcher(line).find())
    .localForEach(System.out::println);
}
```

Listing 7.9: Distributed Stream implementation of the *grep* workload.

7.6.2 Sort

The *sort* workload reads an input text file, sorts the lines and outputs the result. Since the `sorted` operation has been reimplemented in the framework to support larger-than-memory datasets, the programmer does not have to be concerned about memory usage during the sorting phase. Thus, sorting a stream of data simply uses the `sorted` operation, as shown in listing 7.10.

After sorting, the results are added to another Distributed Collection. In this workload, the resulting Distributed Collection is a `HDFSStringCollectionWriter`, thus data that is `add`ed to the collection is written to a HDFS file.

7.6.3 Word-count

The *word-count* workload outputs the number of occurrences of each word in an input text file. The Distributed Stream algorithm used is the same as the detailed word-count described in listing 7.5.

```
static void sort(DistributedCollection<String> lines,
  DistributedCollection<String> result)
{
  lines
    .stream()
    .sorted()
    .localForEach(result::add);
}
```

Listing 7.10: Distributed Stream implementation of the *sort* workload.

7.6.4 Bayes

The *Bayes* workload classifies lines of text from an input file using a naive Bayes classifier. Before the workload, the classifier is trained with a subset of the input file, and the models are stored in a HDFS file. Each model contains a map of (*word, probability*) pairs. The elapsed time and resources used in training are not taken into account.

The classification algorithm used is shown in listing 7.11 and is adapted from the Spark implementation. The Hadoop implementation uses the Mahout [7] library. The algorithm is highly parallelisable, and assuming that the models fit in memory, each line is processed in parallel as follows:

1. The line is parsed to obtain a set of words (i.e. duplicates are removed).
2. Iterating through each model, multiply the probabilities of each word (as given in the model) together to obtain the *a posteriori* probability.
3. From these, pick the model which gives the highest *a posteriori* probability for the line.

7.6.5 Connected components

The *connected components* workload finds the subsets of a graph described by an input file that are connected together (directly or indirectly) by edges. The algorithm used in the Distributed Stream version is as follows:

1. Each node reads its local part of the graph and finds the sets of vertices in the subgraph that are connected.
2. All sets of vertices are sent to the first node, which accumulates and merges them if there are overlaps.

The Hadoop and Spark implementations use the Pegasus [20] and GraphX [10] libraries respectively.

7.6.6 PageRank

The *PageRank* workload executes a naive PageRank analysis on an input file describing a directed graph, with each line describing an edge in the graph. The implementation uses an iterative MapReduce approach with a damping factor d of 0.85 as follows:

```
static void classify(DistributedCollection<String> lines,
  DistributedCollection<String> out, String trainFile)
{
  ...
  FileSystem fs = ...; // Used for opening HDFS files
  Pattern delim = Pattern.compile("\\s+");
  // Read models into memory
  Map<String, Map<String, Double>> models;
  ObjectInputStream in = new ObjectInputStream(fs.open(new Path(trainFile)));
  models = (Map<String, Map<String, Double>>) in.readObject();
  ...
  // Run classifier
  lines
    .parallelStream()
    .map(line -> {
      // Get words in line with no duplicates
      Set<String> words = Arrays.stream(delim.split(line)).collect(Collectors.toSet());
      double maxp = -1.0;
      String result = "default";
      // Iterate through each model and compute most probable model
      for (Map.Entry<String, Map<String, Double>> m: models.entrySet())
      {
        double p = 1.0;
        Map<String, Double> model = m.getValue();
        // Compute a posteriori probability
        for (String word: words)
          p *= model.getOrDefault(word, 0.0001);
        // Update most probable model for line
        if (p > maxp)
        {
          maxp = p;
          result = m.getKey();
        }
      }
      return result + "␣" + line;
    })
    .localForEach(line -> {
      out.add(line);
    });
}
```

Listing 7.11: Distributed Stream classification algorithm for the Bayes workload.

1. The PageRank of each node is initialised to 1.

2. Map stage: each source node's PageRank is divided into fragments of equal value to be distributed to each destination node that is linked to from the source node.

3. Reduce stage: for each destination node, the incoming fragment values are summed up and the result r is used to compute the node's new PageRank (which is $rd+1-d$).

4. Repeat steps 2 and 3 for a total of 10 iterations[2].

The Hadoop and Spark implementations were also run for at most 10 iterations.

7.7 Results and discussion

All experimental results from the tests described in section 7.6 can be found in appendix A arranged according to workload. (Several of the graphs are reproduced in this section for convenience.)

Note that even though the tests were carried out as fairly as possible, Distributed Streams are a proof-of-concept and are not heavily optimised. On the other hand, Hadoop and Spark have been under active development for several years and are thus much more optimised. Fault tolerance is also not implemented in Distributed Streams. Therefore, it is not burdened by such overhead. However, fault tolerance overheads may still exist in Hadoop and Spark regardless of the reliability of the clusters.

7.7.1 Execution time

The Distributed Stream implementations of disk-intensive-only workloads (*Bayes*, *grep* and *word-count*) ran significantly faster than or comparable to those for Hadoop and Spark. Figure 7.7 shows the execution time results for the *grep* workload on cluster 2b.

On cluster 1, the performance of Distributed Stream *sort* workloads (which are both disk-intensive and network-intensive) were comparable to Spark's and Hadoop's respectively. However, on clusters 2a and 2b, the *sort* workload ran significantly slower (see figure 7.8 for *sort* workload execution time results on cluster 2b). This was due to the sorting implementation not making full use of the faster network interconnect and available memory. An improved version was implemented with the following changes:

- The data items are sorted locally and partitioned according to the sampled data.

- On each node, several threads are created (one for each compute node except itself) and associated with each partition destined for a non-local node.

- Each thread sends data from its partition to the destination node in sorted order. This replaces the existing implementation which uses a single thread to send all data unsorted.

[2]In general, PageRank algorithms terminate when the new PageRank values differ by less than a threshold. However, due to the different algorithms used, the number of iterations would be different, thus an upper limit for the number of iterations was specified.

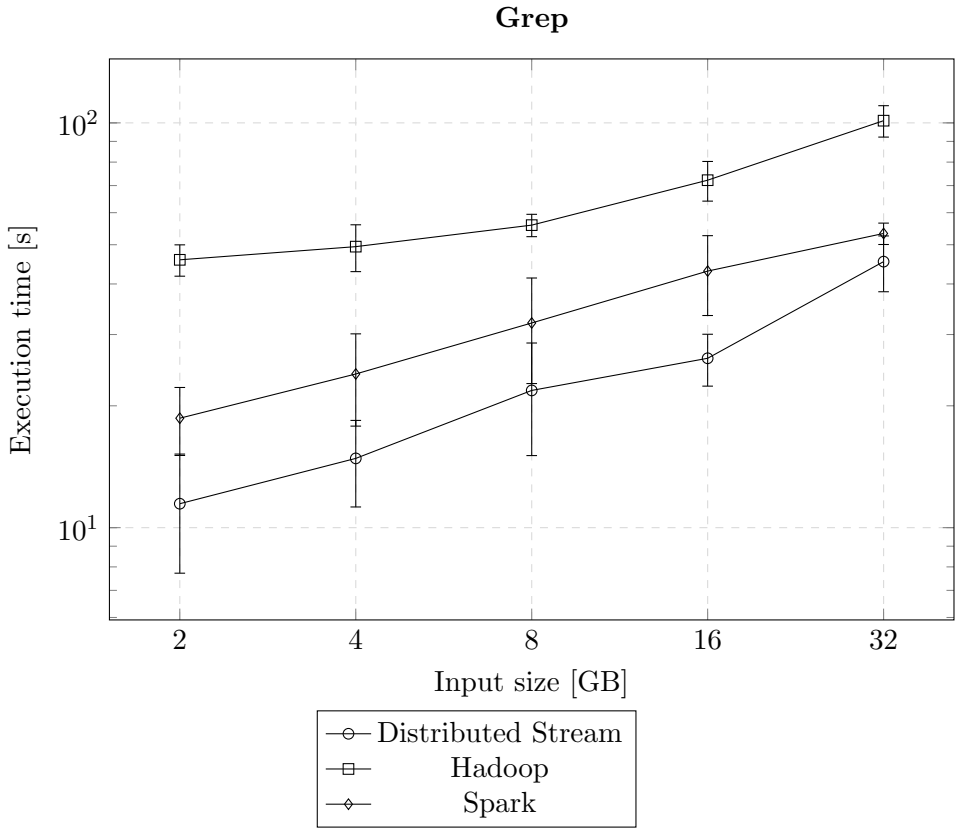


Figure 7.7: Average execution time for the *Grep* workload on cluster 2b (20 nodes).

- Each node receives and merges the incoming data on the fly. This replaces the existing external merge sort as there is sufficient memory in clusters 2a and 2b to hold all the data.

This was substituted for the existing version and measured to run about 1.2× faster than Spark for the largest input size. It appears that the performance of operations requiring communication are dependent on the internals of the transport layer used. In this case, the use of multiple threads for sending data and a single thread for receiving data was faster on MPJ Express than other threading combinations. Thus, modifying the implementation to sort data first allows concurrent sending of data which potentially speeds up execution time.

For graph workloads (*connected components* and *PageRank*), execution times for Distributed Stream implementations vary widely depending on the input size. See figure 7.9 for *PageRank* workload execution time results on cluster 2b. Besides being less optimised, another reason is that Hadoop and Spark have optimised graph processing frameworks and libraries (Pegasus and GraphX) that are closely tied to the respective programming models while the graph algorithms for Distributed Stream implementations were written from scratch. It is likely that if more efficient graph algorithms and optimisations are used, they can scale as well as Hadoop and Spark for larger datasets.

The standard deviations on cluster 1 are generally smaller than those on clusters 2a and 2b. This is because full and exclusive access was given to cluster 1 for the tests, while the nodes in clusters 2a and 2b were part of a larger shared cluster with a job submission mechanism. This hindered efforts to minimise variability in clusters 2a and 2b. Such

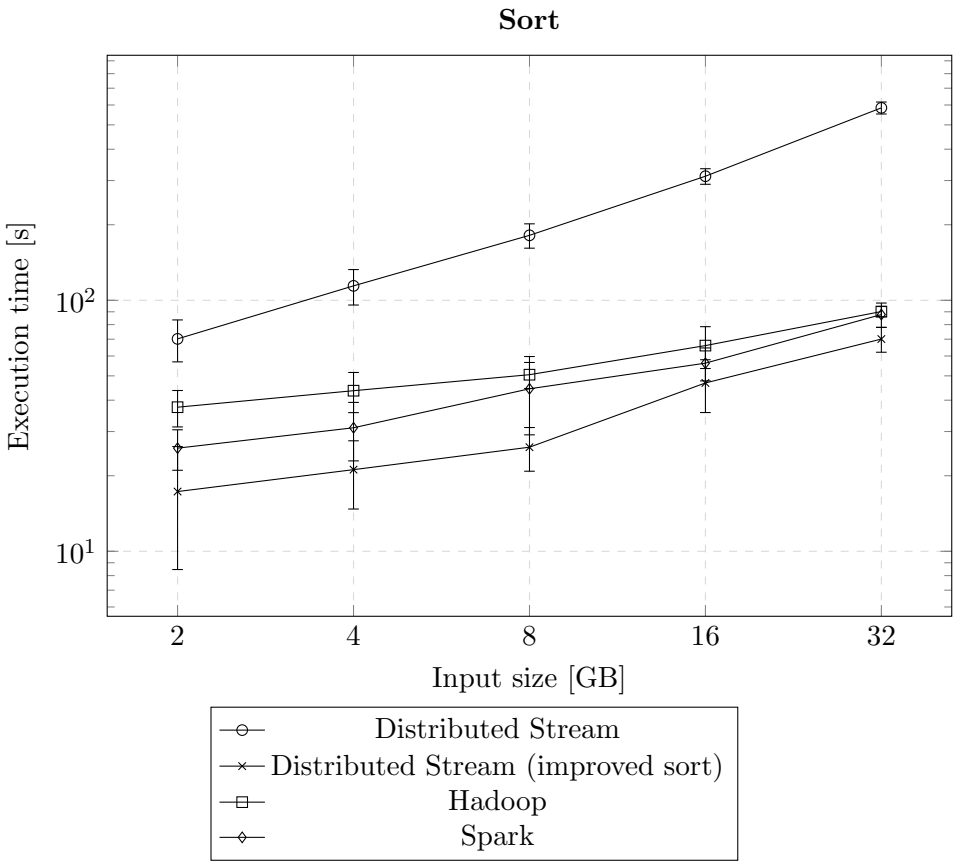


Figure 7.8: Average execution time for the *Sort* workload on cluster 2b (20 nodes).

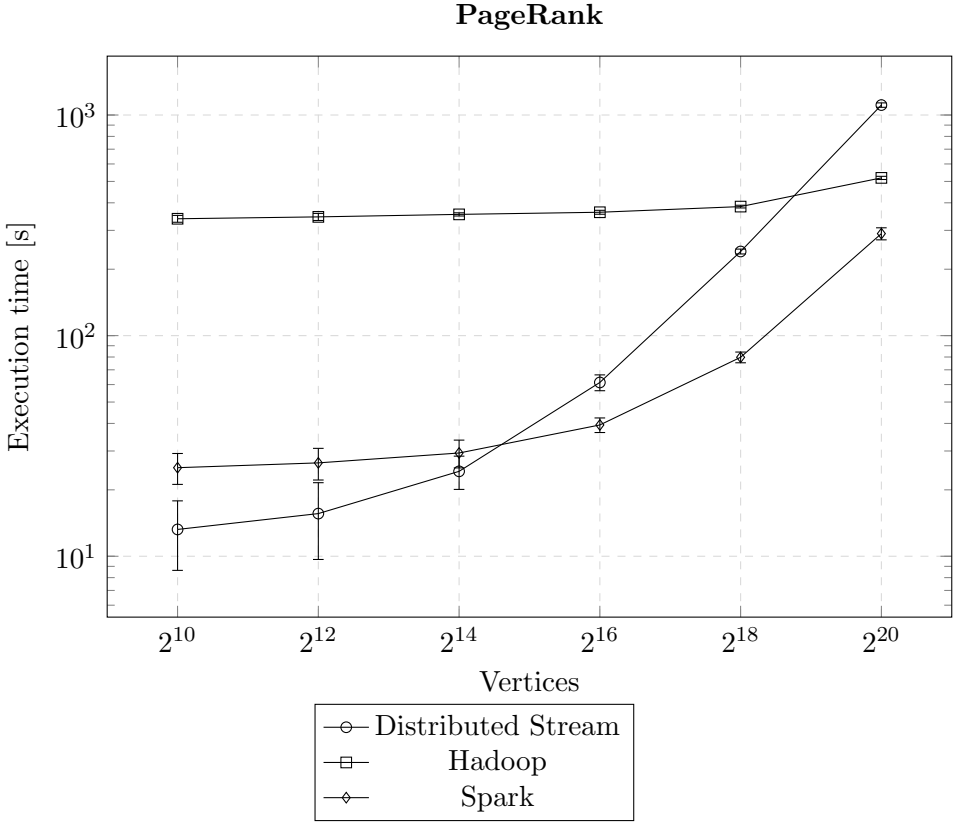


Figure 7.9: Average execution time for the *PageRank* workload on cluster 2b (20 nodes).

factors included interference from other users of the cluster, the lack of hard disk data persistence between jobs and the differences in hardware (especially hard disk I/O speeds).

7.7.2 Network usage

Hadoop and Spark attempt to load-balance tasks across the cluster, which may require nodes to read data on a non-local disk and have it sent across the network. Distributed Streams does not do this, except for sending data items that straddle HDFS blocks. This can be seen in results for the *grep* and *word-count* workloads, with network usage being higher and less predictable for Hadoop and Spark than Distributed Streams (see figure 7.10 for the *grep* workload network usage results on cluster 2b). In particular, on clusters 2a and 2b, some of the standard deviations for Spark's network usage were larger than the values themselves. These large variations are likely caused by the difference in HDFS block locations in the input data (the input data could not be persistent on these clusters and thus had to be copied to HDFS for every run, resulting in different block placements).

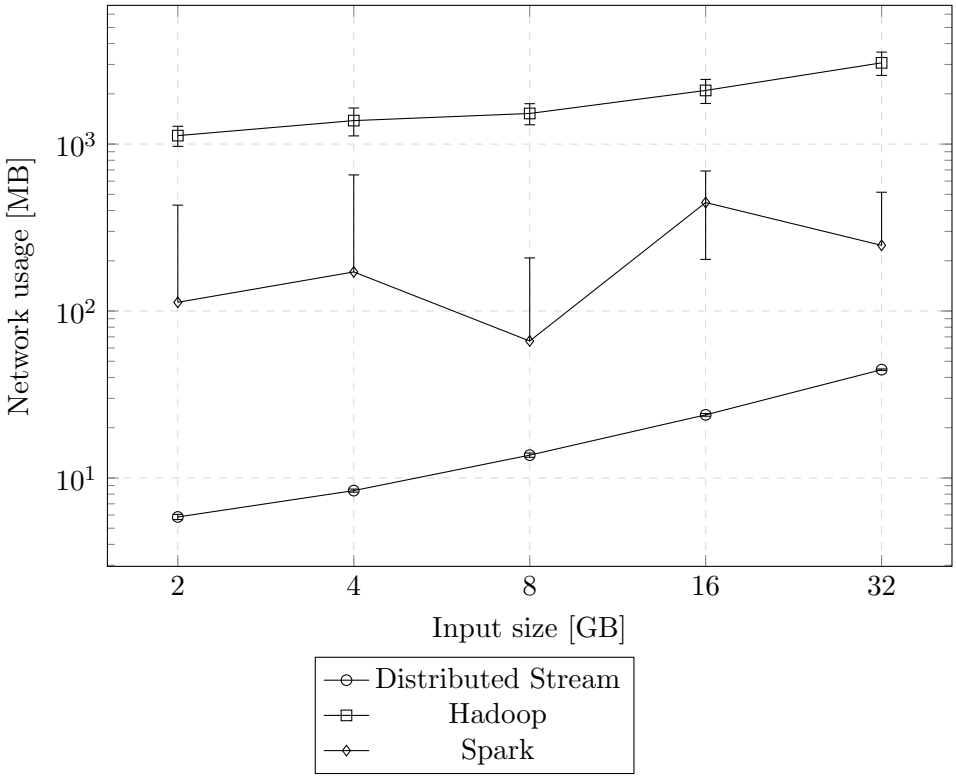


Figure 7.10: Average network usage for the *Grep* workload on cluster 2b (20 nodes).

The network usage patterns are generally determined by what each workload does. Low usage is observed in disk-intensive workloads *grep* and *word-count* which output a small volume of data (see figure 7.10). Higher usage is observed in the *Bayes* workload due to writing each line of input to output together with its classification and attempts by HDFS to distribute the output data over the cluster. The difference in *Bayes* results between Hadoop and the other frameworks is likely due to the less optimised HDFS bindings for Distributed Streams (`HDFSStringCollectionWriter`) and Spark.

The highest usage is found in the *sort* workload, which is due to the shuffling and

subsequent writing out of the entire dataset (which HDFS tries to balance over the cluster). Again, Hadoop has the least usage due to its optimised shuffling and sorting. See figure 7.11 for network usage of the *sort* workload on cluster 2b.

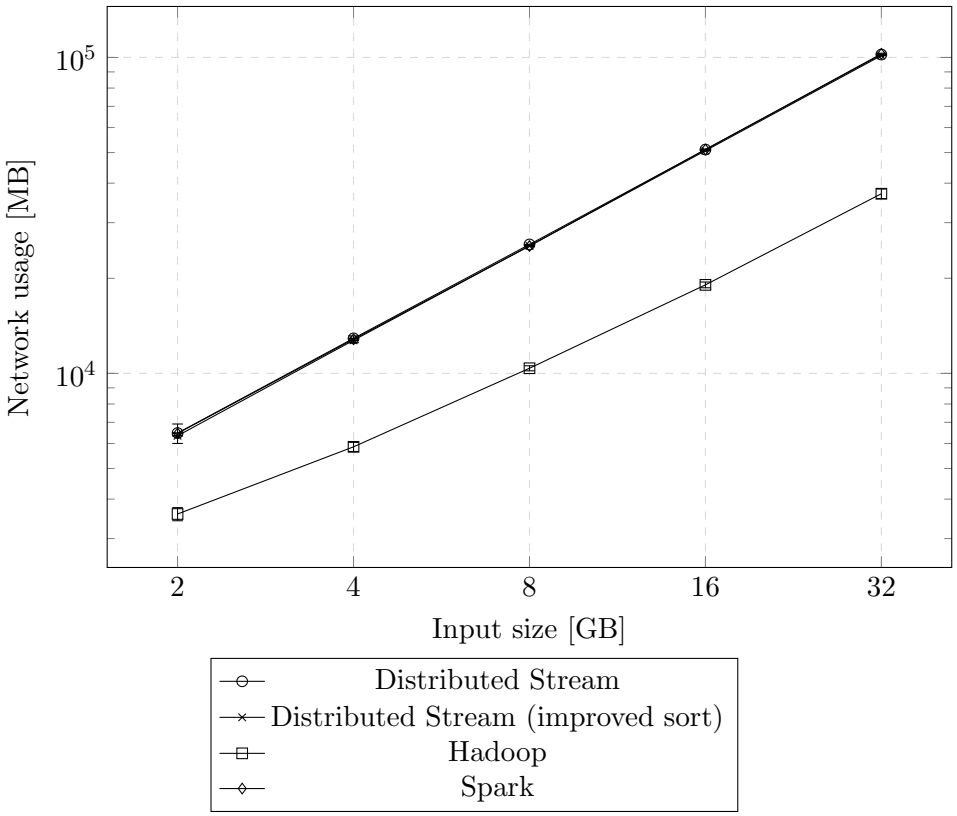


Figure 7.11: Average network usage for the *Sort* workload on cluster 2b (20 nodes).

Usage for the *connected components* and *PageRank* workloads are dependent on the input data size, approaching linear growth as the dataset gets larger. Since the algorithms of these two workloads vary widely by framework, they exhibit very different usage patterns. See figure 7.12 for network usage of the *PageRank* workload on cluster 2b.

As previously mentioned, Distributed Streams are not heavily optimised. This is especially true of the communication layer. Though network usage of Distributed Streams was generally in the high end, they still ran as fast as (if not faster than) the other frameworks for non-graph workloads. This highlights the efficiency of Distributed Streams, and especially the Java 8 Stream framework that it builds upon.

7.7.3 Further evaluation

More experiments were carried out to determine the cause of the different execution times observed in section 7.7.1. The *grep* workloads for each framework were re-run on cluster 2b with a script that outputs the disk usage statistics per second on each compute node as recorded by the Linux kernel for the duration of the computation. Then, each run's disk read per second statistics were obtained by summing up the individual node's statistics.

Figure 7.13 shows the disk reads per second of one run from each framework, which is representative of the other runs. Although all frameworks read from the same HDFS

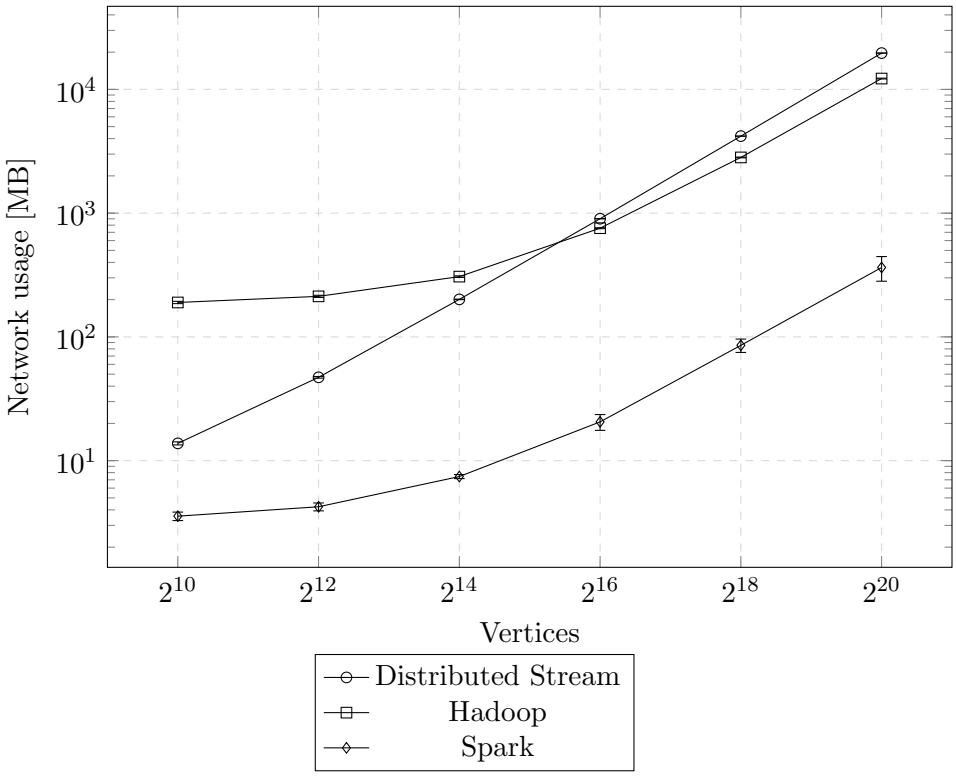


Figure 7.12: Average network usage for the *PageRank* workload on cluster 2b (20 nodes).

dataset, Distributed Streams had the highest disk read throughput and spent the least amount of time with disks idle. Thus, it was more efficient in reading data from disk and computation than the other frameworks. A possible reason is that the `HDFSStringCollection` implementation only allows one thread to read from disk at a time, reducing contention and enabling mostly sequential disk reads. This was shown in section 4.2.2 to be faster than allowing multiple threads access to the disk at the same time, which degenerates into random disk reads.

7.8 Summary

This section recapitulates the evaluation findings and brings up general observations.

7.8.1 Expressive power

The Distributed Stream framework is expressive enough to implement Big Data algorithms across a wide range of domains. However, algorithms involving graphs (and other high-level concepts) tend to be verbose. This is because, unlike Spark, the framework has no special-purpose operations to simplify graph programming. In Hadoop's case, its wide usage has prompted the development of libraries, such as Mahout, that map well-known Big Data algorithms to MapReduce jobs. In addition, Java 8 does not have graph data structures in its core libraries. Thus the programmer will need to use a third-party library, implement their own, or use distributed versions of Java collections and maps to achieve the same results.

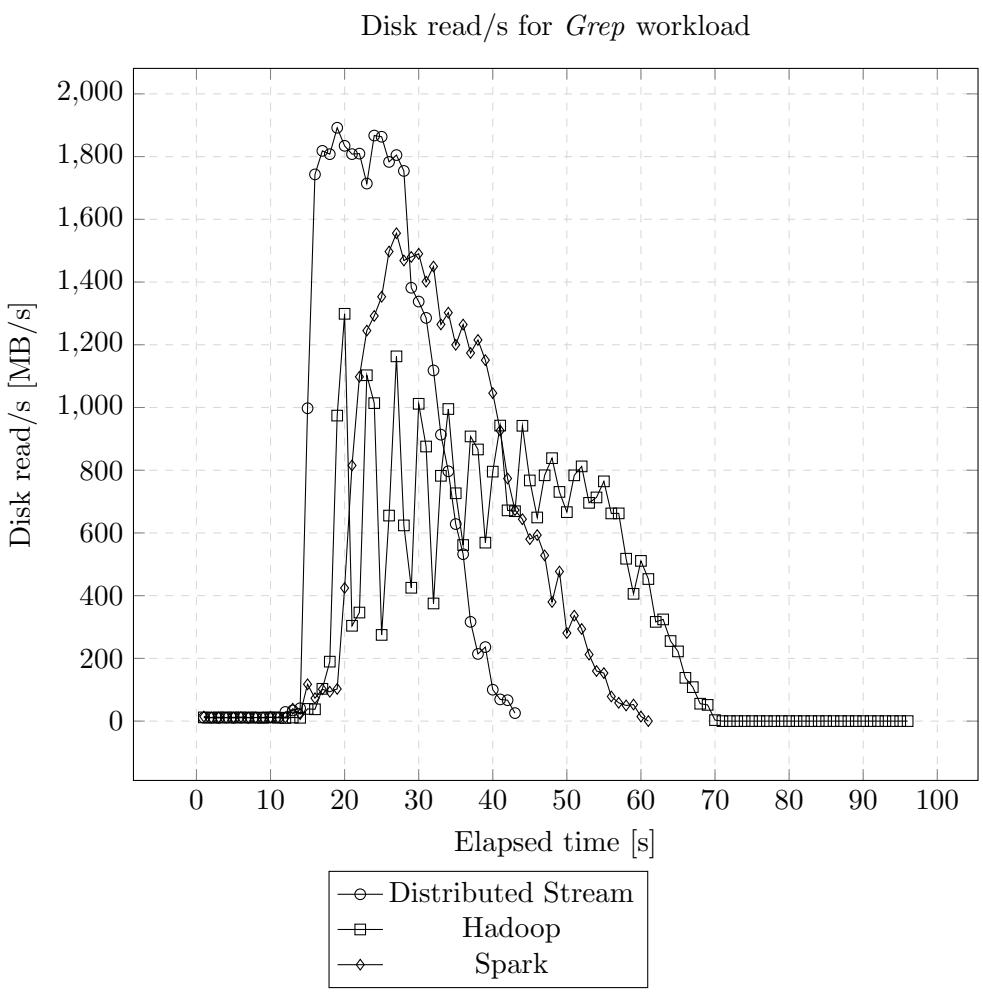


Figure 7.13: Disk reads per second for the *Grep* workload on cluster 2b (20 nodes).

7.8.2 Performance

Of the six workloads, Distributed Streams performed better or comparably well with three workloads (*grep*, *word-count* and*Bayes*). One of the workloads (*sort*) achieved better performance with an improved implementation on clusters 2a and 2b. Distributed Streams did not scale well on the remaining workloads (*connected components* and *PageRank*, which are graph workloads) due to the algorithms used and the unoptimised communication layer. With an increase in cluster size, the execution time and network usage of Distributed Streams scaled as well as those of Hadoop and Spark.

Currently, Distributed Streams do not provide any fault tolerance above that which is already provided by the underlying communication layer implementation. Yet, these results illustrate that as a thin layer, the Distributed Streams API is a suitable extension to Java 8 Streams for Big Data computations.

With Distributed Streams being proof-of-concept, there is room for efficiency improvements. To support larger clusters, reliability issues will also need to be addressed while not significantly degrading performance.

Chapter 8

Conclusion

This thesis has proposed, implemented and evaluated Distributed Streams, a framework for Big Data computation based on Java 8 Streams. Returning to the hypothesis stated in section 1.2:

> This thesis contends that the Java 8 Stream framework is inadequate to support all the requirements of programming Big Data systems. However, it is possible to extend the framework to meet all the requirements for programming Big Data systems, and still achieve performance comparable to or exceeding those of the popular Big Data frameworks (Hadoop and Spark, for example). Furthermore, this can be achieved in a way which retains compatibility with existing Java 8 software.

This has been adequately demonstrated in the thesis.

The thesis establishes in chapter 1 that although Java is the base language for widely used Big Data frameworks, it does not have Big Data capabilities on its own, though recent changes in Java 8 have brought it a step closer. The introduction of Streams and lambda expressions in Java 8 have allowed more concise implementations of data-parallel programs. However, current Big Data applications need the computing power and storage capacity of multiple computers, so these new features are still insufficient for Big Data processing on their own. Thus, this chapter supports the first sentence of the thesis hypothesis:

> This thesis contends that the Java 8 Stream framework is inadequate to support all the requirements of programming Big Data systems.

In chapter 2, the thesis reviewed a number of Big Data programming models and frameworks. Many of the popular Big Data frameworks (e.g. Hadoop, Spark, Storm and S4) introduce programming models that require programmers to significantly rewrite their programs and grasp new concepts in order to use them effectively. Applicability and expressiveness is sometimes reduced due to restrictions in the programming model (e.g. only mappers and reducers allowed in Hadoop/MapReduce computations, and Storm topologies run indefinitely). Some of the frameworks (e.g. Hive, Shark and Ricardo) bring Big Data capabilities to existing languages without significantly changing the existing programming model. However, these mainly cater to high-level languages and, in the case

of Ricardo, introduce another query language in the process. Java 8 Streams provide a pipeline-based model, which is similar to Spark's programming model, but is restricted to a single JVM.

From these findings, chapter 3 derived a list of requirements for an extended Java 8 Stream framework that addresses Big Data issues to:

- Support running programs on a cluster of nodes.
- Support large datasets (both single-node and distributed).
- Maintain data locality within a node and across the cluster.
- Provide operations for distributing data and computation across the cluster.
- Be a drop-in replacement for Java 8 Streams.

These requirements define what needs to be met in order to support the following clause in the thesis hypothesis:

> However, it is possible to extend the framework to meet all the requirements for programming Big Data systems...

Chapter 4 introduced the concept of Stored Collections as a data source for Java 8 Streams to allow efficient reading of data on disk and thus support large datasets on a single node. Preliminary evaluations of Stored Collections used in conjunction with Java 8 Streams show better performance compared to Hadoop and that data locality is preserved within a node.

Chapter 5 introduced the main contribution of the thesis, the Distributed Stream framework that satisfies the remainder of the requirements. It extends the Java 8 Stream framework to multiple JVMs, allowing it to be used in a cluster for greater parallelism. Computation over the cluster is made possible by the concept of compute nodes and groups, which allow addressing and grouping of nodes. A Distributed Stream is the distributed analog of a Java 8 Stream, using a pipeline that is replicated over the compute nodes. New operations have been added to distribute data and computation across compute nodes, as well as local operations that are useful in optimising the performance of these pipelines. Also part of the framework are Distributed Collections, which are a data source for Distributed Streams and encapsulate data that is partitioned across the cluster. Furthermore, data from a distributed source can be fed into a pipeline that was written for Java 8 Streams, demonstrating backward compatibility. A proof-of-concept implementation was described in chapter 6.

The framework was evaluated in chapter 7 for programmability and performance against popular Big Data frameworks. Programmability-wise, the framework is expressive enough for MapReduce and streaming computations. Performance-wise, on many workloads the framework either ran as fast as or faster than Hadoop and Spark. Hence, chapters 4 to 7 fully support the second sentence of the thesis hypothesis:

> However, it is possible to extend the framework to meet all the requirements for programming Big Data systems, and still achieve performance comparable

to or exceeding those of the popular Big Data frameworks (Hadoop and Spark, for example).

In the remainder of this chapter, section 8.1 discusses areas for improvement and further research, and section 8.2 concludes the thesis.

8.1 Future work

Chapter 7 has shown the advantages in using Distributed Streams. However, it has also highlighted a number of limitations (as presented and implemented in the thesis). This section discusses possible improvements in programmability and performance, and ways to adapt the framework to a wider range of uses.

8.1.1 Fault tolerance

The Distributed Stream framework as implemented in chapter 6 is a proof-of-concept and has no fault tolerance other than that provided by the underlying communication layer. This is one of the Big Data attributes listed in section 3.1 not addressed in the implementation. This is generally not a problem for small clusters consisting of recent commodity hardware.

In environments with large clusters, the chance of hardware failure increases and it is important that these failures do not prevent long-running computations from finishing. Therefore, fault tolerance from node failures is an essential requirement in these situations. (The software is assumed to be fault-free, so fault tolerance from software failures are ignored.)

This section considers two ways to add fault tolerance to Distributed Streams – by extending the current (SPMD-based) programming model, or by changing the programming paradigm to be more similar to existing Big Data frameworks (master-slave). In the current model, all compute nodes run the same program which synchronises the execution of all nodes. Hence, the same pipeline would be running on all participating compute nodes. In a hypothetical model that uses master-slave, the program would be run on a master node which hands parts of pipelines to compute nodes for execution. The compute nodes would then communicate among themselves to pass data around where necessary.

SPMD

As mentioned in section 2.4.1, using redundant nodes and checkpointing are two ways fault tolerance on SPMD can be achieved. This can be introduced into the model by extending the `ComputeGroup` class to handle failed nodes. Thus, compute groups would monitor nodes for any failures and rerun the affected computations on other nodes. Using this method, different levels and implementations of fault tolerance can be used in a single program if needed.

There are several MPI implementations with extensions that address fault tolerance. Two of them are FT-MPI [33] and StarFish MPI [2]. The drop in performance for both approaches were shown to be minimal, so they can be considered for porting to Java and use in Distributed Streams.

Master-slave

Alternatively, one can base the fault tolerance implementation on those used in existing Big Data frameworks.

In Hadoop, fault tolerance is achieved by the jobtracker periodically sending heartbeat messages to the tasktrackers and restarting lost tasks on other nodes if no reply is received. The Distributed Stream framework can adopt a similar approach by making the following changes to the programming model:

- Running the main program on the master node only, and having the compute nodes receive work from the master. Thus, the master node can send the same piece of work to another compute node if one fails.
- Disallowing addressing of specific compute nodes and compute groups. Thus, all Distributed Stream pipelines will be replicated across the whole cluster. Hints may be given to the framework to specify the number of destination nodes a `distribute` operation should send data to, but the actual number cannot be guaranteed. This may also imply having a process that load-balances work on the cluster to avoid situations where most work is sent to one node (unless explicitly hinted).

In the implementation, the master node would keep track of computations occurring on the compute nodes and, on failure, restart them on another node. During `distribute` operations, data sent over the network is logged, in case it needs to be re-sent to another node.

This approach requires more changes to the programming model and implementation, but the resulting fault tolerance mechanism would resemble that of other Big Data approaches more closely. The new programming model would be simpler (as there are no compute nodes or groups) but with reduced expressiveness. Data parallelism and data locality, which are important in Big Data computations, would be preserved but task parallelism would be eliminated, as the entire cluster would be performing a single stream computation. This is unlikely to be a significant issue as most programs are not expected to use the full expressiveness of the current model. The performance of this approach can also be compared with that of the current approach by running the same workloads on the new implementation.

8.1.2 Supporting different data formats

Another Big Data attribute listed in section 3.1 is the support of different data formats. Currently, support only exists for accessing HDFS files (through `HDFSStringCollection`s that were implemented for evaluating the framework). A comprehensive implementation of Distributed Streams would include Distributed Collections that read from other common data formats such as Cassandra and HBase [5].

8.1.3 Optimising performance

The framework's specification is sufficiently broad to allow for many dissimilar implementations, so an optimised one may look very different from the current prototype. An optimised implementation will likely have the following properties:

- Uses a fast, low-level communication layer. This may be accomplished by using a network layer that offers Java Native Interface (JNI) support. Thus, part of its implementation would bypass the JVM and potentially reduce latencies for sending and receiving messages.

- Minimises the overhead of partitioning data by keeping the number of partitions small. However, too small a number may unnecessarily delay processing and consume memory while data is buffered, so some benchmarking and heuristics may be needed to arrive at a reasonable range of values.

- Writes temporary data to disk only when memory is low. Certain complex operations (for example `sorted`) will benefit from this optimisation if there is enough memory to hold the temporary data, saving costly disk I/O.

- If accurate results are not required, operations which consume memory proportional to the input size (such as `distinct`) can be reimplemented to use approximate algorithms, or have a counterpart that uses these. This would lower the program's memory usage.

8.1.4 Programming extensions

Chapter 7 described two graph workloads (*connected components* and *PageRank*) that were used in the evaluation of Distributed Streams. Since the framework does not have operations that simplify graph programming, implementations of these workloads manipulated primitive data in Distributed Collections. Although this gives the programmer more control over the implementation, it leads to more verbose code. To address this issue, further research can be carried out to determine common patterns that are found in graph algorithms that use Distributed Streams, and then consider extending the framework in one of a few ways:

- Extend the `DistributedStream` interface to handle streams of vertices and/or edges and include graph operations.

- Implement a separate graph library that uses Distributed Streams for computation.

8.1.5 Improving the prototype implementation

The prototype implementation as presented in chapter 6 suffers from a number of limitations. The following suggests ways to work around them:

- The current implementation does not allow null values to be used in streams, as it is used internally as the last data item sent from a compute node during a `distribute` operation. A possible solution is to send a *more-data* indicator together with each item, so that a destination node knows whether to expect more data items from a source node.

- The current implementation only allows objects that are serialisable to be sent during a `distribute` operation. This is a limitation of using the MPJ Express `MPI.OBJECT`

type. As a workaround, serialisation can be done using an alternative library (for example Kryo [56]) and the resulting byte stream sent as a `MPI.BYTE` array. Thus, objects that do not implement `Serializable` can still be serialised and used in `distribute` operations without any changes, improving compatibility with existing code.

The procedure for starting programs can also be improved by hiding MPJ Express-specific and other implementation details. For example, to run a program in the `HelloWorld.class` file, the startup command could be simplified to that in listing 8.1.

```
$DSTREAM_PATH/run [-c cluster-spec] HelloWorld [args...]
```

Listing 8.1: Improved command to run a Distributed Stream program (in class file `HelloWorld.class`). The `$DSTREAM_PATH` environment variable contains the installed path of the framework. The `-c cluster-spec` option describes the cluster the program will run on.

8.1.6 Evaluation of simultaneous workloads

All tests in chapter 7 consisted of single workloads that were run across the whole cluster. This is useful for gauging the baseline performance of frameworks. However, clusters may be running several Big Data computations at once and it would be insightful to observe the performance of Distributed Streams on multiple simultaneous workloads. (The BigDataBench benchmarks also do not consider such cases.) The performance of these tests may depend largely on how efficiently data is read from disk, as there potentially would be multiple tasks needing disk access at the same time.

There could also be variations of these tests, including running different workloads (for example *sort* and *word-count*) and starting them at different times. These would more closely simulate Big Data clusters that respond to queries from front-end servers.

8.1.7 Evaluation of other use cases

Although chapter 7 compared the programming models of Distributed Streams and Storm, there was no evaluation of Storm's performance. In part, this is due to the difference in purposes of each framework – Storm is designed for low-latency event processing. The BigDataBench suite also did not include workloads for Storm. However, it would be interesting to compare the performance of both frameworks on event processing-based workloads.

Currently, each worker thread in Distributed Streams receives a batch of data items every time they call the `trySplit` method of a spliterator. Although this reduces overheads for Big Data processing, this may increase the latency of responses when the rate of arrival of data items (events) is slow. In this case, the spliterator implementation can be modified to reduce the batch size of data items, possibly to 1 if a near-immediate response is needed.

8.2 Closing remarks

Java 8 Streams is an efficient framework for data parallel computation within a JVM. This can be seen in results from experiments in chapter 4 to determine the single-node performance of Java 8 Streams. Today, however, Big Data involves distributed computing which Java 8 Streams does not provide. This thesis has presented in chapter 5 an extended framework, called Distributed Streams, that addresses the Big Data shortcomings in Java 8 Streams. The framework is also backward compatible with Java 8 Streams, allowing most existing programs to be incrementally changed to use the full functionality of Distributed Streams.

This framework was evaluated in chapter 7 and many workloads were shown to be faster or as fast using Distributed Streams compared to Hadoop and Spark. Given more resources, it should be possible for performance to be improved further.

The thesis has taken a bottom-up approach to introducing Big Data into an existing framework. This approach was taken after having observed that Java 8 Streams already perform efficiently on a single node, and in many cases more efficiently than Hadoop.

More generally, the thesis has shown that extending an existing shared memory programming model to support Big Data is a viable alternative to rewriting the entire program in a new and unfamiliar model. The popular Big Data frameworks such as Hadoop revolutionised Big Data computing, but require programmers to learn a new programming model. The approach this thesis has taken preserves most properties of the existing model as well as its syntax.

Appendix A

Distributed Stream Evaluation Results

This appendix contains all the execution time and network usage results from the Distributed Stream framework evaluation in chapter 7, grouped by workload.

A.1 Grep

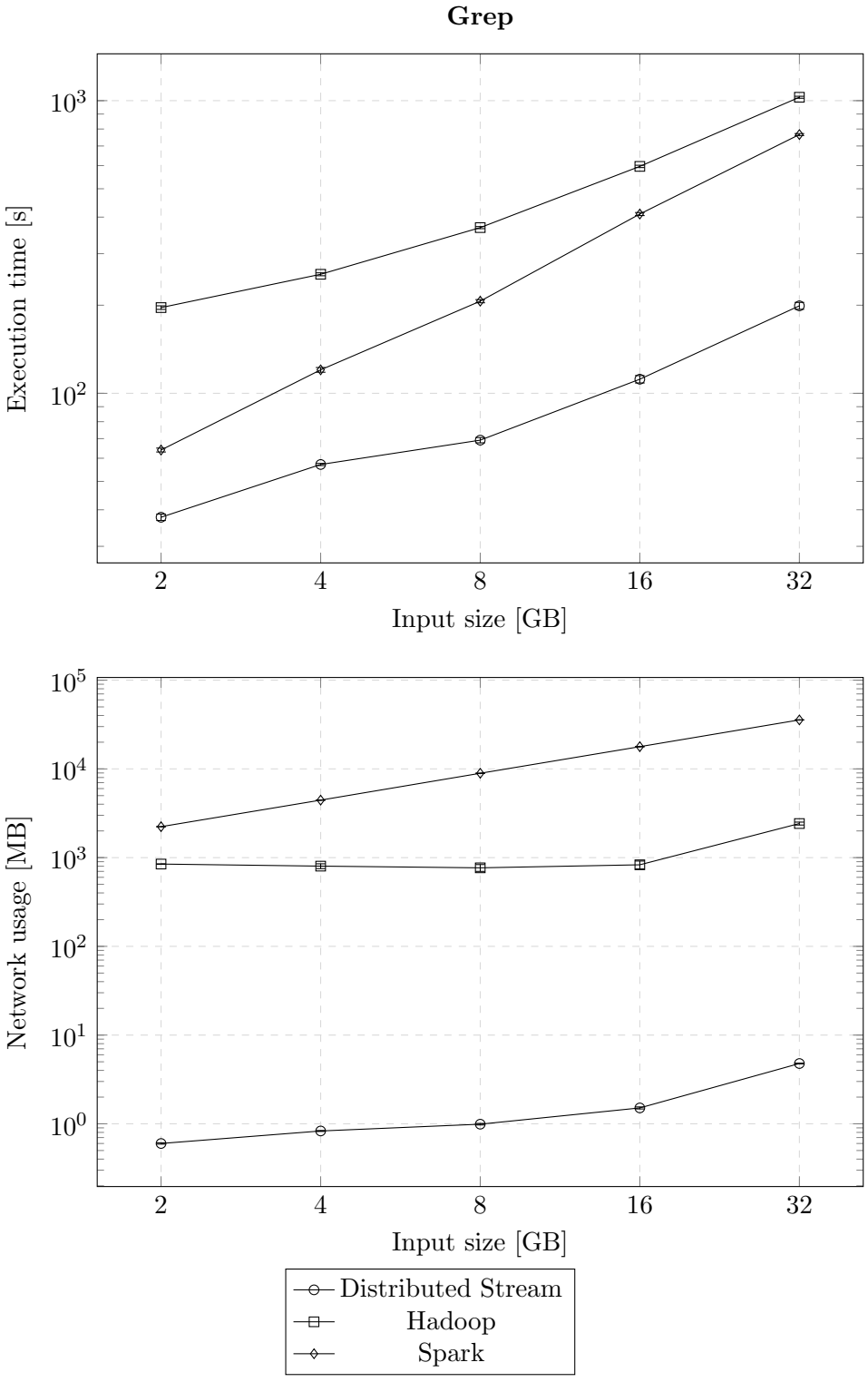


Figure A.1: Average execution time and network usage for the *Grep* workload on cluster 1 (6 nodes).

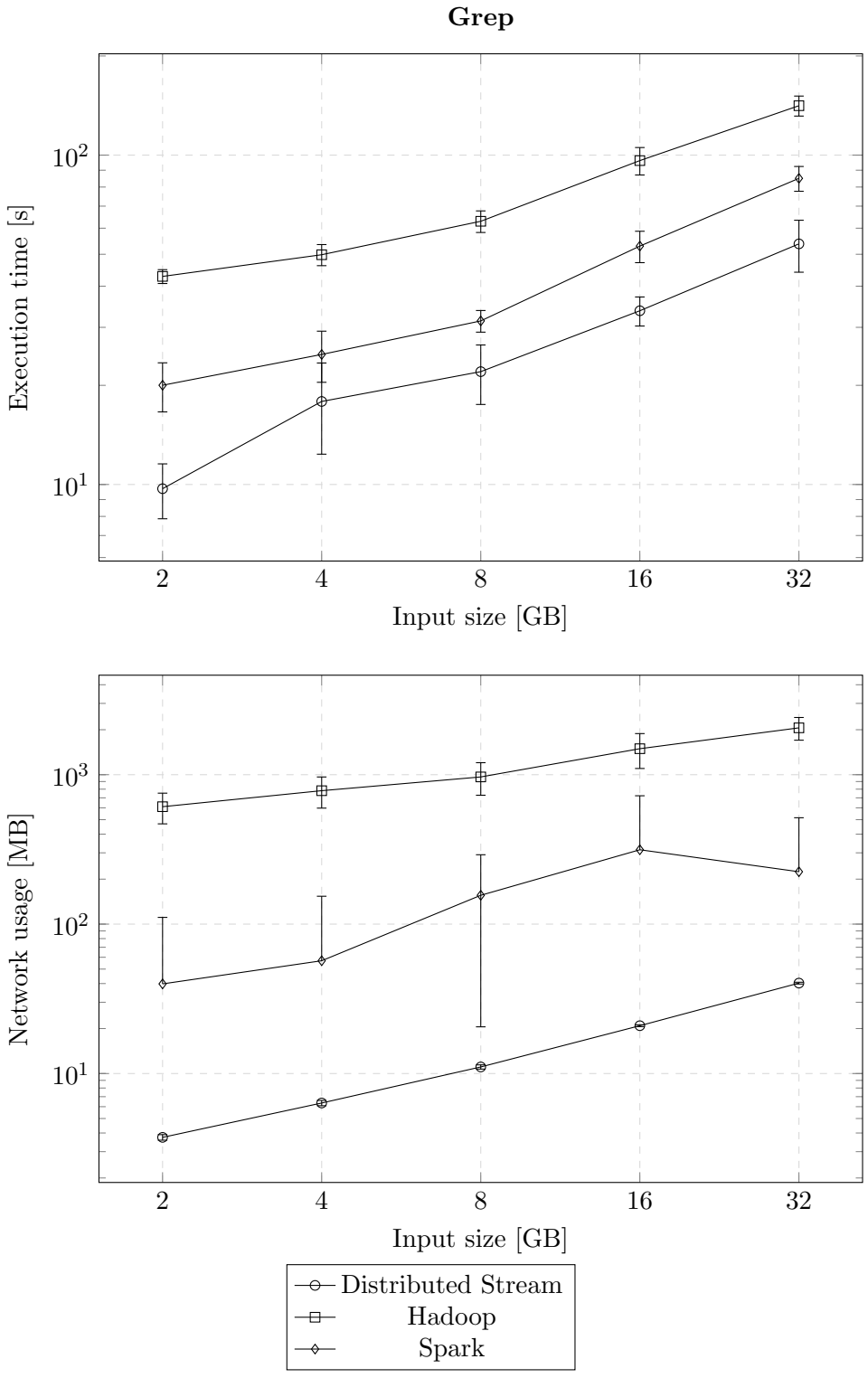


Figure A.2: Average execution time and network usage for the *Grep* workload on cluster 2a (10 nodes).

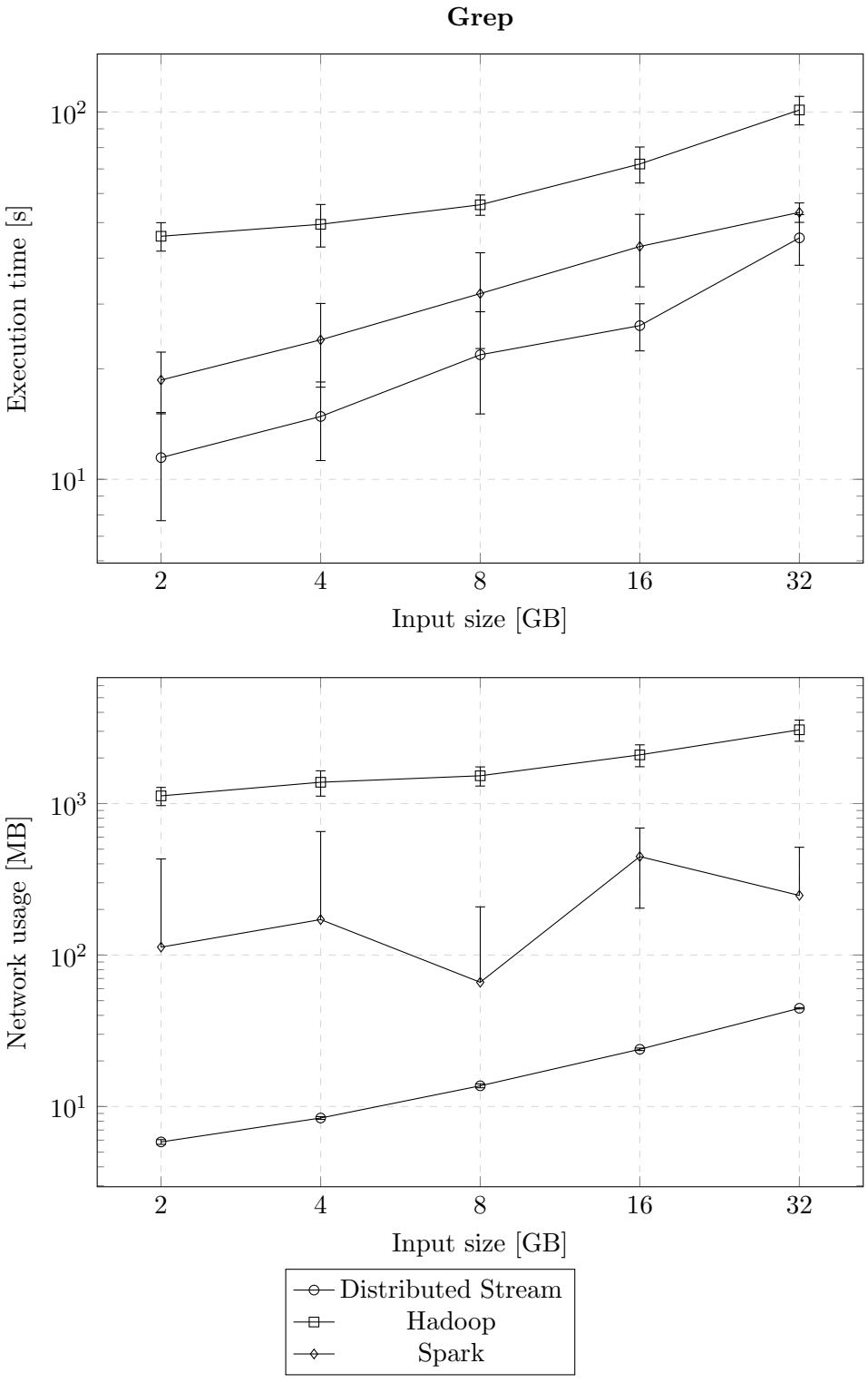


Figure A.3: Average execution time and network usage for the *Grep* workload on cluster 2b (20 nodes).

Grep	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	37.72	0.95	1	0	196.32	2.75	846	6	64.01	0.98	2233	0
4	57.13	0.49	1	0	255.11	2.93	802	50	120.27	2.22	4448	0
8	69.17	1.43	1	0	368.38	3.04	768	68	206.46	2.58	8927	0
16	111.73	3.24	2	0	596.24	5.42	830	85	409.81	4.91	17810	0
32	199.09	5.39	5	0	1027.28	8.00	2422	92	766.04	5.97	35664	1

Table A.1: Average execution time and network usage for the *Grep* workload on cluster 1 (6 nodes).

Grep	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	9.71	1.84	4	0	42.83	2.07	611	142	20.01	3.40	40	71
4	17.87	5.51	6	0	49.84	3.67	782	184	24.82	4.39	57	97
8	22.01	4.51	11	0	62.97	4.74	966	237	31.38	2.37	156	135
16	33.69	3.41	21	0	96.24	9.21	1493	392	52.94	5.79	315	408
32	53.77	9.66	40	1	141.26	9.87	2059	357	85.02	7.35	224	291

Table A.2: Average execution time and network usage for the *Grep* workload on cluster 2a (10 nodes).

Grep	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	11.46	3.74	6	0	45.89	4.08	1124	155	18.64	3.56	113	318
4	14.83	3.58	8	0	49.46	6.55	1383	263	23.97	6.15	172	482
8	21.83	6.76	14	0	55.90	3.55	1526	220	32.04	9.34	66	142
16	26.21	3.84	24	0	72.19	8.11	2096	345	43.04	9.61	447	243
32	45.42	7.15	45	0	101.26	9.02	3068	490	53.31	3.24	247	267

Table A.3: Average execution time and network usage for the *Grep* workload on cluster 2b (20 nodes).

A.2 Sort

On cluster 1, Spark did not complete for the largest input size due to insufficient disk space for the computation.

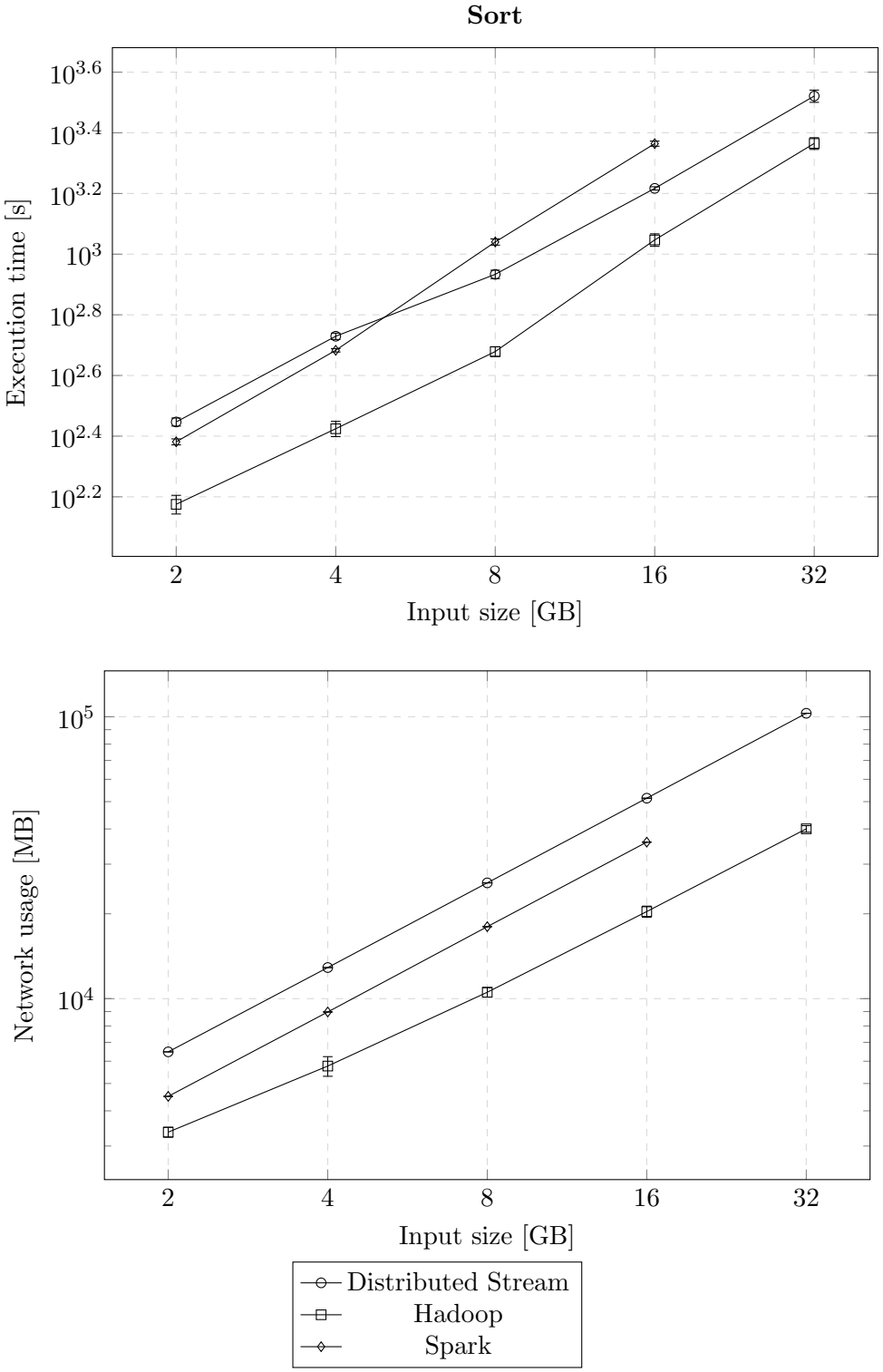


Figure A.4: Average execution time and network usage for the *Sort* workload on cluster 1 (6 nodes).

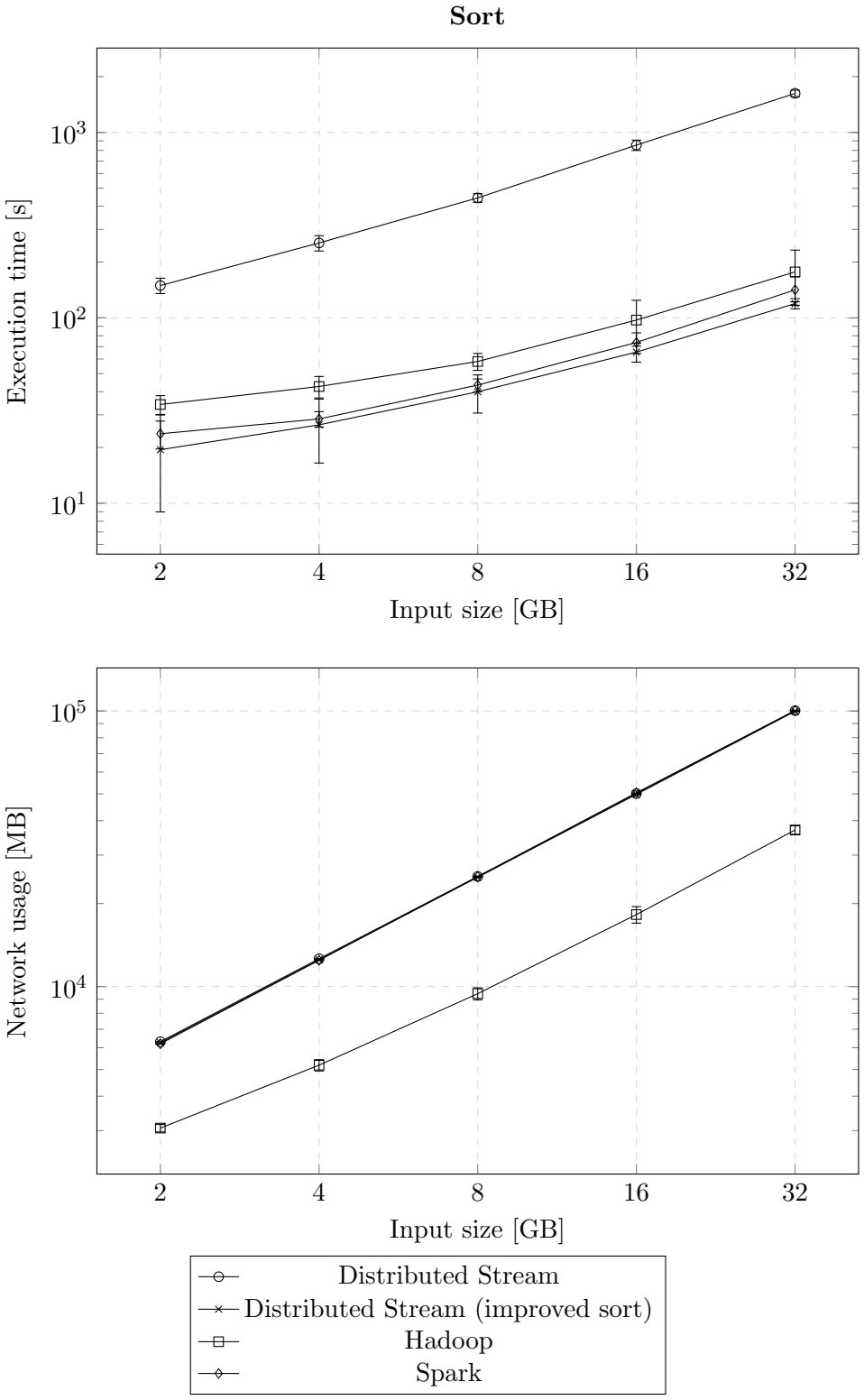


Figure A.5: Average execution time and network usage for the *Sort* workload on cluster 2a (10 nodes).

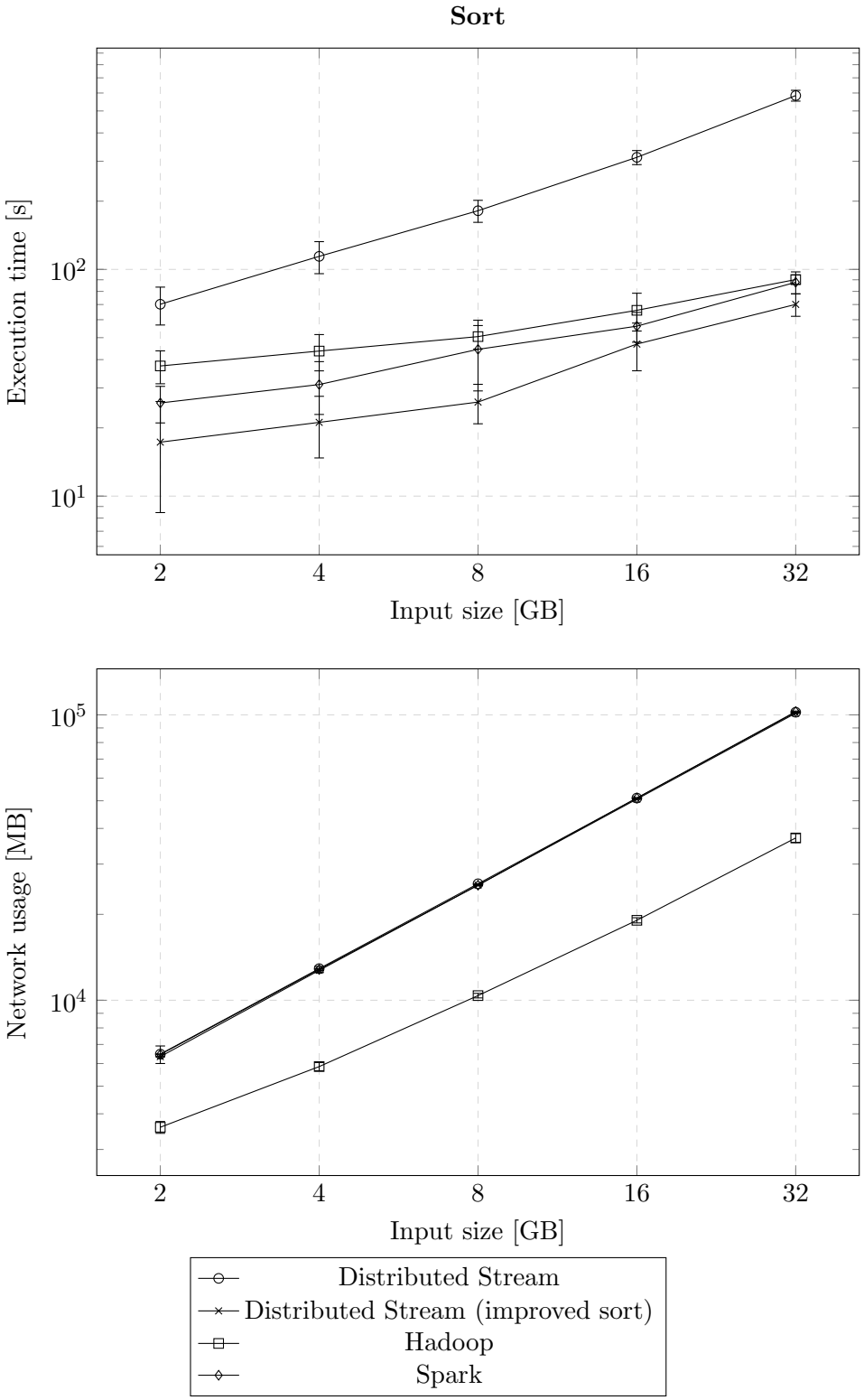


Figure A.6: Average execution time and network usage for the *Sort* workload on cluster 2b (20 nodes).

Sort	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	279.67	8.53	6476	2	149.69	10.60	3361	136	240.55	5.46	4502	0
4	536.14	12.92	12893	2	265.66	15.39	5767	461	481.89	6.16	8969	0
8	857.86	27.07	25752	3	477.29	17.32	10537	406	1096.63	25.86	18005	1
16	1647.31	20.76	51423	7	1113.71	52.79	20370	923	2313.60	45.18	35922	1
32	3320.10	154.05	102894	11	2317.01	102.82	40036	1063				

Table A.4: Average execution time and network usage for the *Sort* workload on cluster 1 (6 nodes).

Sort	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	149.39	14.22	6309	5	34.10	3.96	3064	95	23.70	4.06	6200	6
4	253.51	24.07	12615	11	42.63	5.65	5184	245	28.49	2.71	12454	6
8	443.62	23.94	25088	9	58.17	6.09	9424	473	43.40	3.42	25054	133
16	854.53	54.75	50150	19	97.30	27.03	18250	1262	73.67	9.31	50500	782
32	1624.36	65.64	100288	19	176.96	54.98	37003	1319	141.66	24.71	100409	288

Sort	Distributed Stream (improved sort)			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	19.45	10.47	6254	66
4	26.48	10.00	12500	16
8	39.94	9.29	24917	23
16	65.34	7.64	49823	55
32	119.42	7.60	99725	142

Table A.5: Average execution time and network usage for the *Sort* workload on cluster 2a (10 nodes).

Sort	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	70.20	13.34	6475	10	37.51	6.21	3589	172	25.76	4.73	6460	457
4	114.18	18.41	12907	6	43.64	7.95	5856	194	31.06	8.14	12827	321
8	181.53	20.19	25597	5	50.54	6.01	10382	191	44.41	15.30	25305	103
16	312.19	22.24	51107	7	66.01	12.52	19067	399	56.18	8.43	51120	174
32	584.63	32.04	102152	15	90.21	4.26	37042	1293	87.75	9.78	102650	482

Sort	Distributed Stream (improved sort)			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	17.30	8.84	6346	5
4	21.15	6.41	12729	15
8	25.99	5.14	25319	34
16	46.85	11.13	50677	73
32	70.10	7.99	101360	114

Table A.6: Average execution time and network usage for the *Sort* workload on cluster 2b (20 nodes).

A.3 Word-count

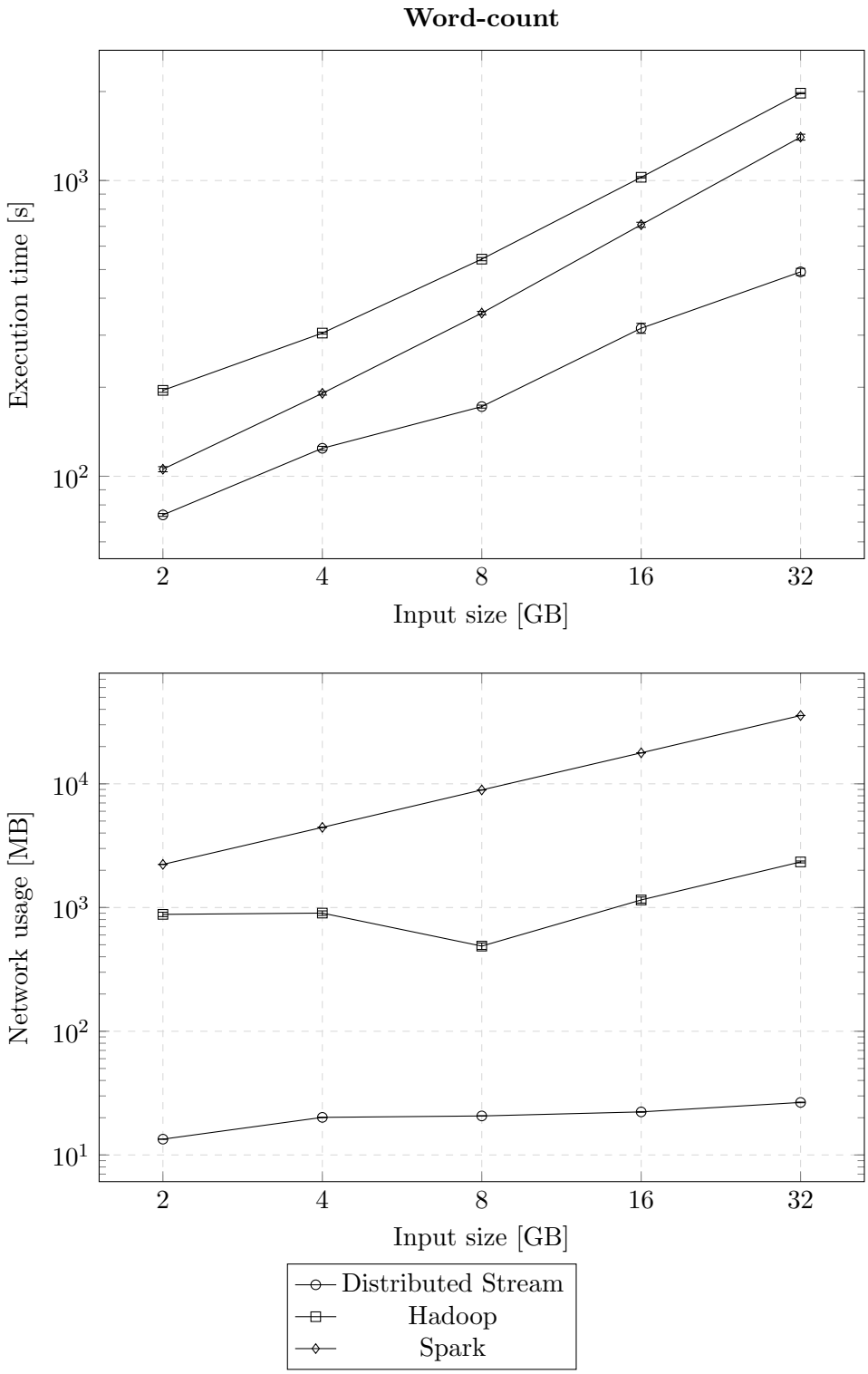


Figure A.7: Average execution time and network usage for the *Word-count* workload on cluster 1 (6 nodes).

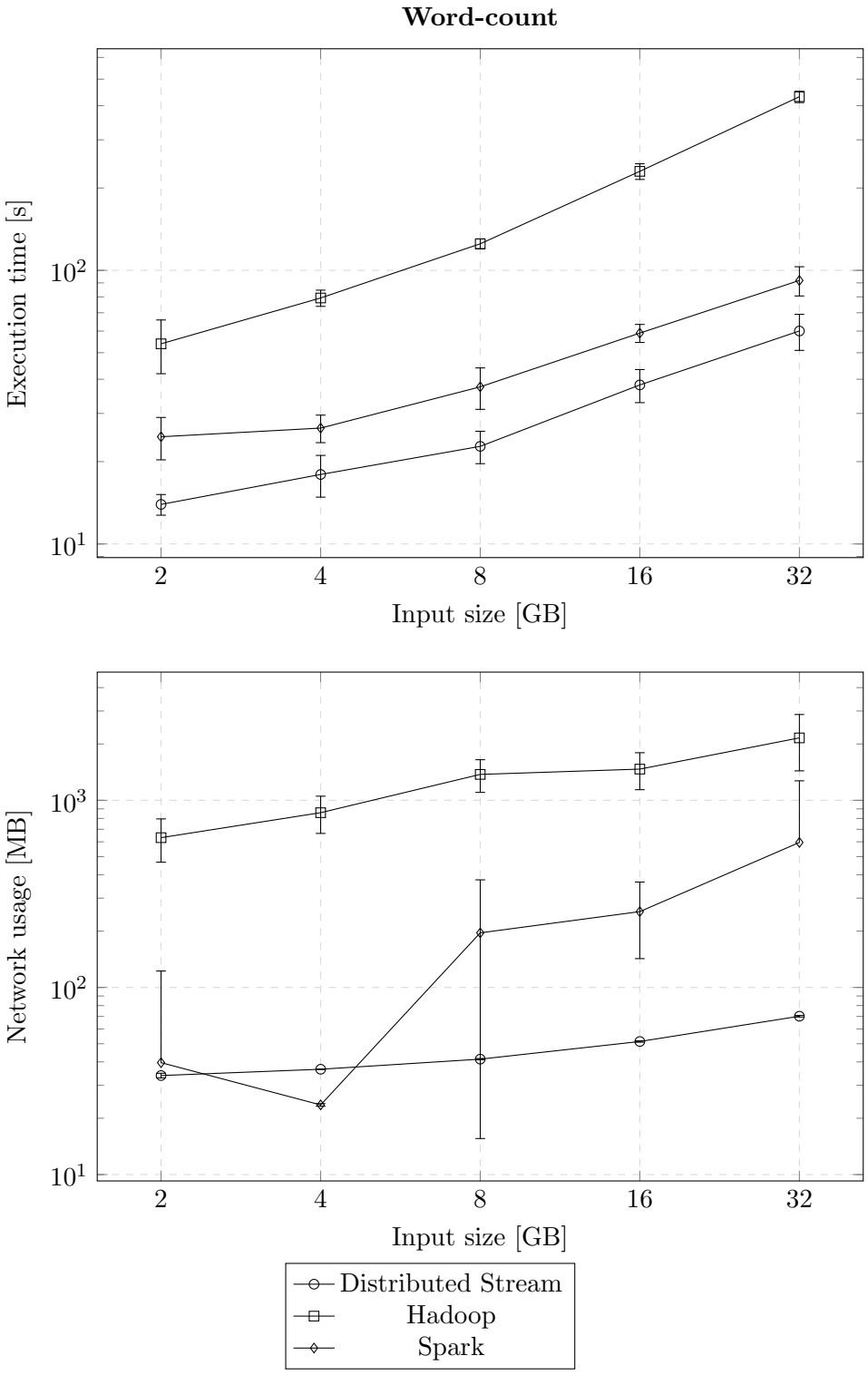


Figure A.8: Average execution time and network usage for the *Word-count* workload on cluster 2a (10 nodes).

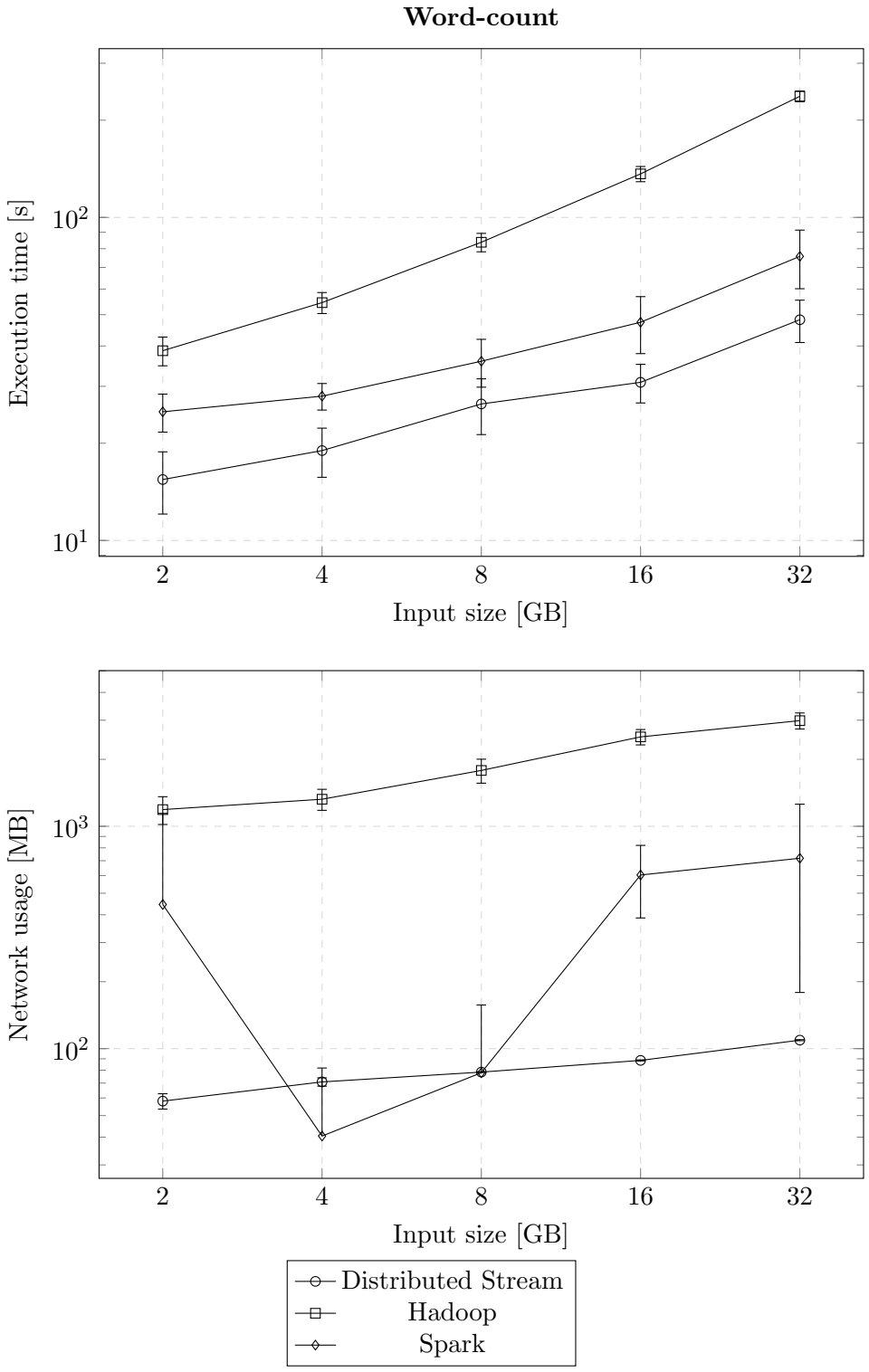


Figure A.9: Average execution time and network usage for the *Word-count* workload on cluster 2b (20 nodes).

Word-count	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	74.01	0.81	13	0	195.23	2.90	880	36	105.66	2.10	2231	0
4	124.33	1.90	20	0	304.93	2.30	902	38	190.84	2.64	4445	0
8	172.03	1.98	21	0	542.38	6.76	488	29	356.34	4.84	8921	0
16	316.66	12.36	22	0	1025.80	5.38	1151	64	708.92	13.79	17797	0
32	491.09	13.39	27	0	1974.53	8.66	2337	41	1402.26	32.19	35637	0

Table A.7: Average execution time and network usage for the *Word-count* workload on cluster 1 (6 nodes).

Word-count	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	13.95	1.21	34	1	53.90	11.99	631	164	24.65	4.35	40	83
4	17.95	3.12	37	0	79.28	5.38	859	193	26.52	3.06	24	0
8	22.74	3.08	41	0	125.07	5.27	1375	273	37.53	6.48	196	180
16	38.13	5.27	51	0	230.11	15.14	1468	328	58.99	4.47	254	112
32	60.04	9.04	70	1	431.15	20.68	2155	718	91.87	11.24	595	676

Table A.8: Average execution time and network usage for the *Word-count* workload on cluster 2a (10 nodes).

Word-count	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2	15.44	3.37	58	5	38.68	3.95	1189	170	25.02	3.37	445	698
4	18.98	3.29	71	3	54.48	4.07	1323	143	27.97	2.64	40	41
8	26.46	5.19	78	0	83.77	5.49	1783	220	35.88	6.06	78	79
16	30.87	4.22	89	0	136.37	7.38	2524	201	47.37	9.49	604	217
32	48.22	7.22	109	1	237.15	9.11	2987	247	75.76	15.60	718	539

Table A.9: Average execution time and network usage for the *Word-count* workload on cluster 2b (20 nodes).

A.4 Bayes

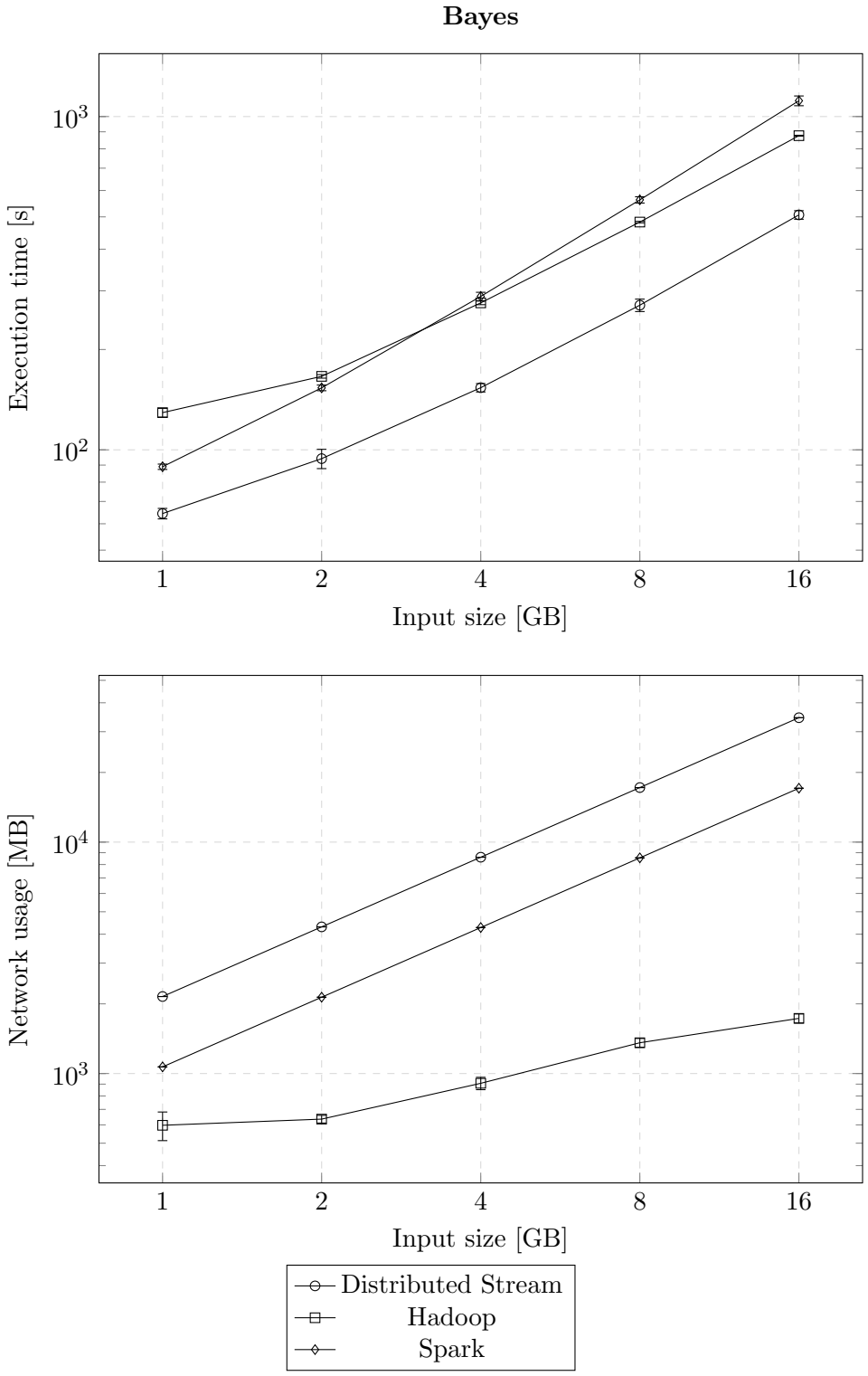


Figure A.10: Average execution time and network usage for the *Bayes* workload on cluster 1 (6 nodes).

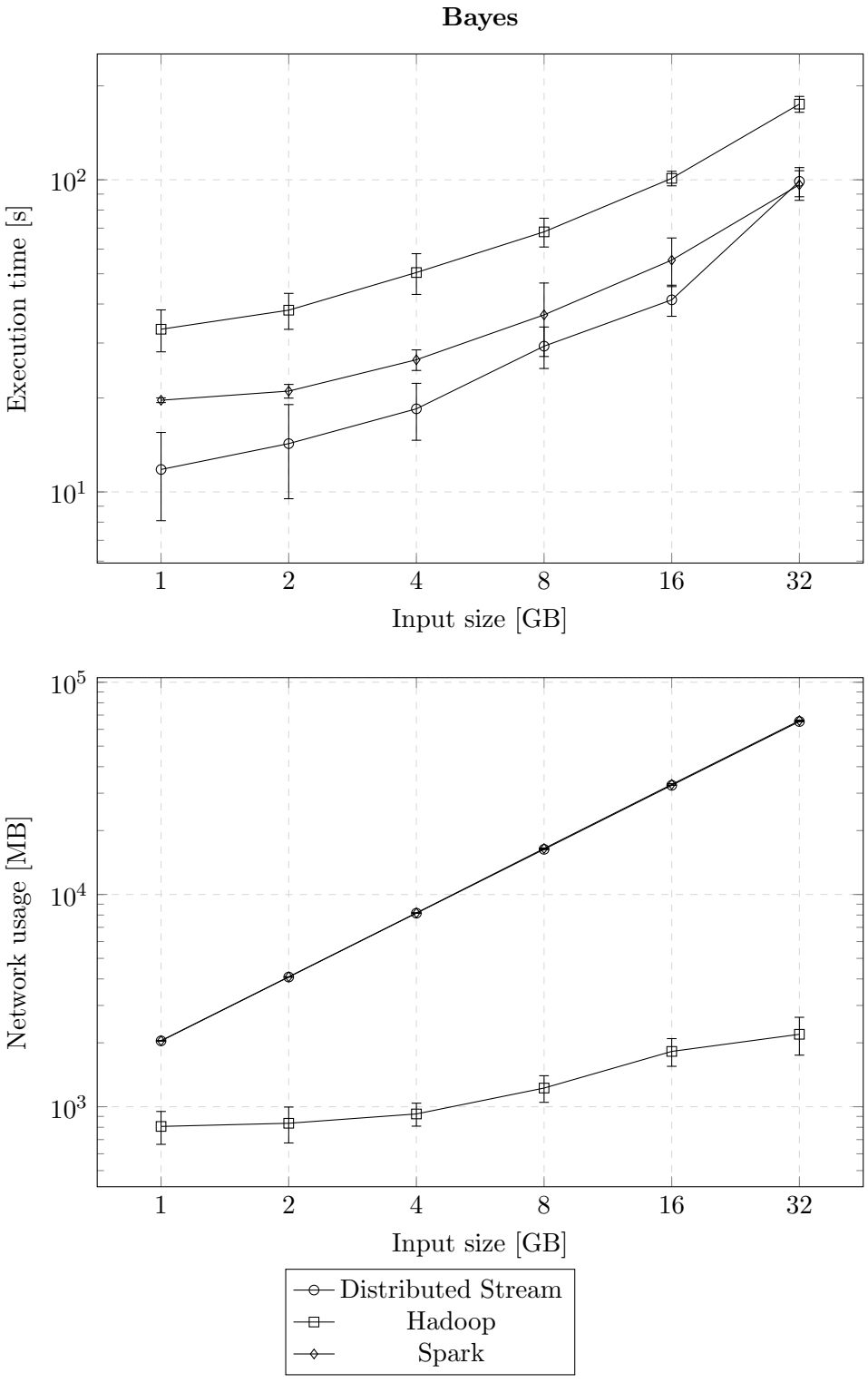


Figure A.11: Average execution time and network usage for the *Bayes* workload on cluster 2a (10 nodes).

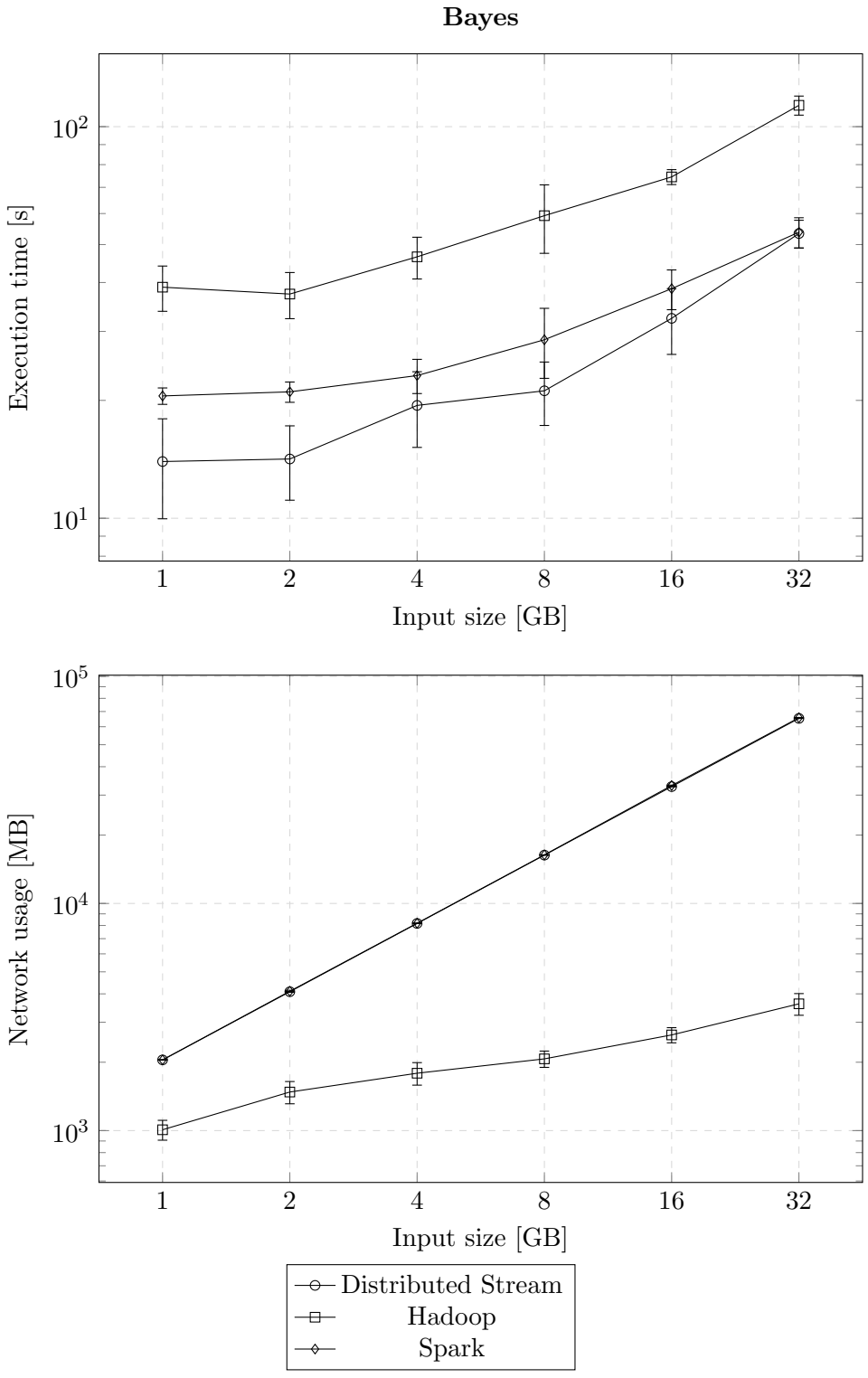


Figure A.12: Average execution time and network usage for the *Bayes* workload on cluster 2b (20 nodes).

Bayes	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
1	64.39	2.33	2152	1	129.30	3.87	598	85	88.98	1.64	1069	1
2	94.10	6.26	4303	1	166.13	2.01	635	24	153.45	2.99	2137	1
4	153.62	4.43	8604	1	276.02	3.45	907	56	288.63	8.35	4274	1
8	271.64	11.65	17206	2	482.38	2.52	1358	59	562.40	12.16	8549	3
16	506.92	14.96	34479	3	876.19	2.72	1731	77	1115.32	37.54	17093	4

Table A.10: Average execution time and network usage for the *Bayes* workload on cluster 1 (6 nodes).

Bayes	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
1	11.80	3.71	2043	0	33.21	5.10	807	142	19.67	0.34	2047	1
2	14.28	4.77	4083	0	38.23	5.04	836	161	21.04	1.06	4092	1
4	18.46	3.82	8162	0	50.46	7.53	925	115	26.51	2.01	8184	1
8	29.31	4.45	16320	1	68.13	7.21	1224	176	36.93	9.77	16464	103
16	41.25	4.71	32686	31	101.09	5.42	1822	272	55.30	9.82	33000	312
32	98.86	10.59	65282	6	174.84	10.23	2194	444	96.46	10.50	65928	406

Table A.11: Average execution time and network usage for the *Bayes* workload on cluster 2a (10 nodes).

Bayes	Distributed Stream				Hadoop				Spark			
Input (GB)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
1	13.96	3.99	2048	0	38.92	5.15	1009	101	20.53	0.99	2048	1
2	14.17	3.05	4087	0	37.38	5.04	1479	166	21.03	1.25	4107	41
4	19.42	4.25	8166	0	46.52	5.68	1789	203	23.14	2.32	8184	1
8	21.16	3.90	16324	1	59.26	11.77	2069	171	28.57	5.80	16368	2
16	32.40	6.19	32698	28	74.43	3.27	2637	202	38.58	4.49	33052	160
32	53.35	4.33	65285	1	113.38	6.35	3615	395	53.76	4.78	65705	200

Table A.12: Average execution time and network usage for the *Bayes* workload on cluster 2b (20 nodes).

A.5 Connected components

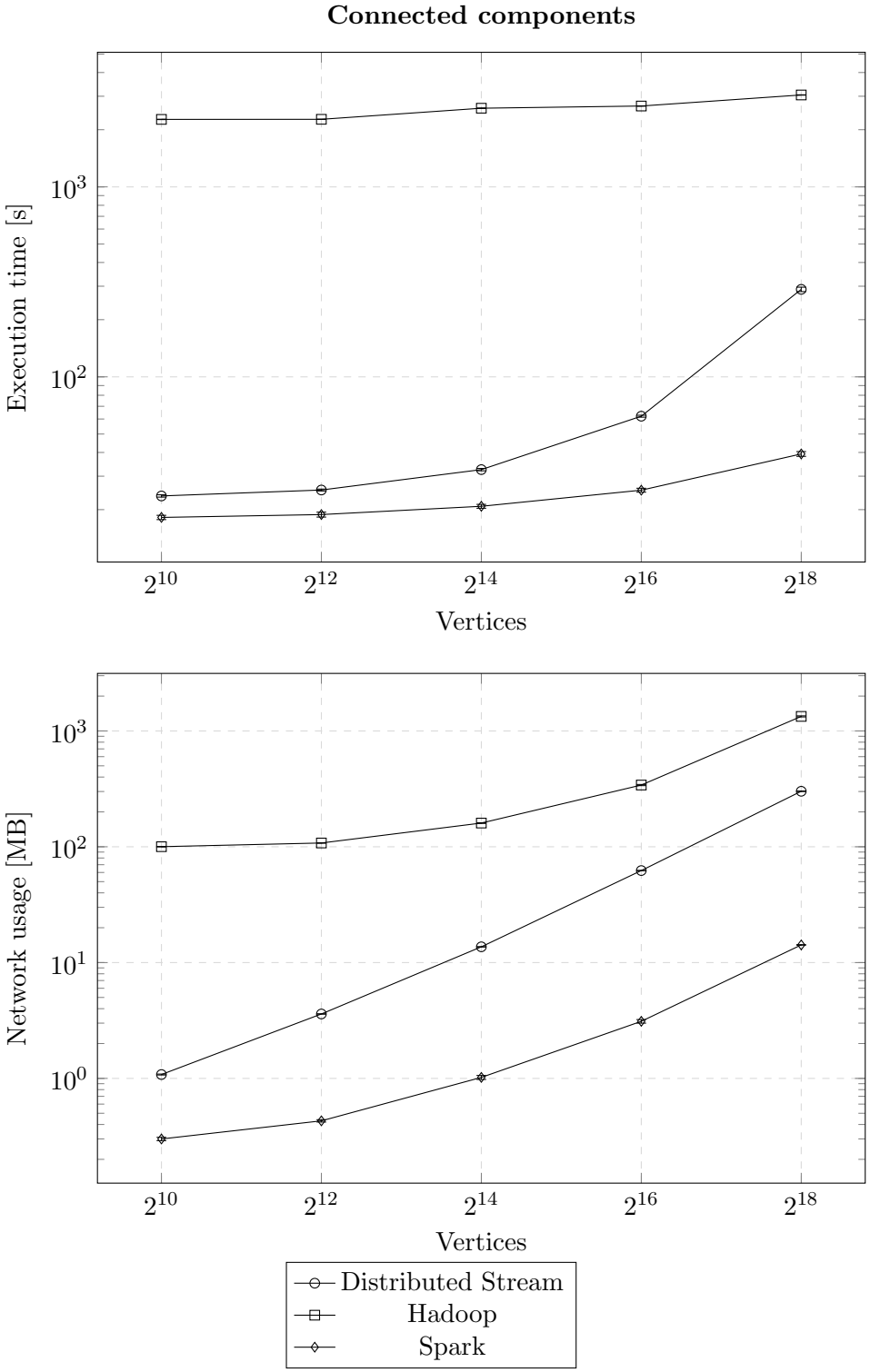


Figure A.13: Average execution time and network usage for the *Connected components* workload on cluster 1 (6 nodes).

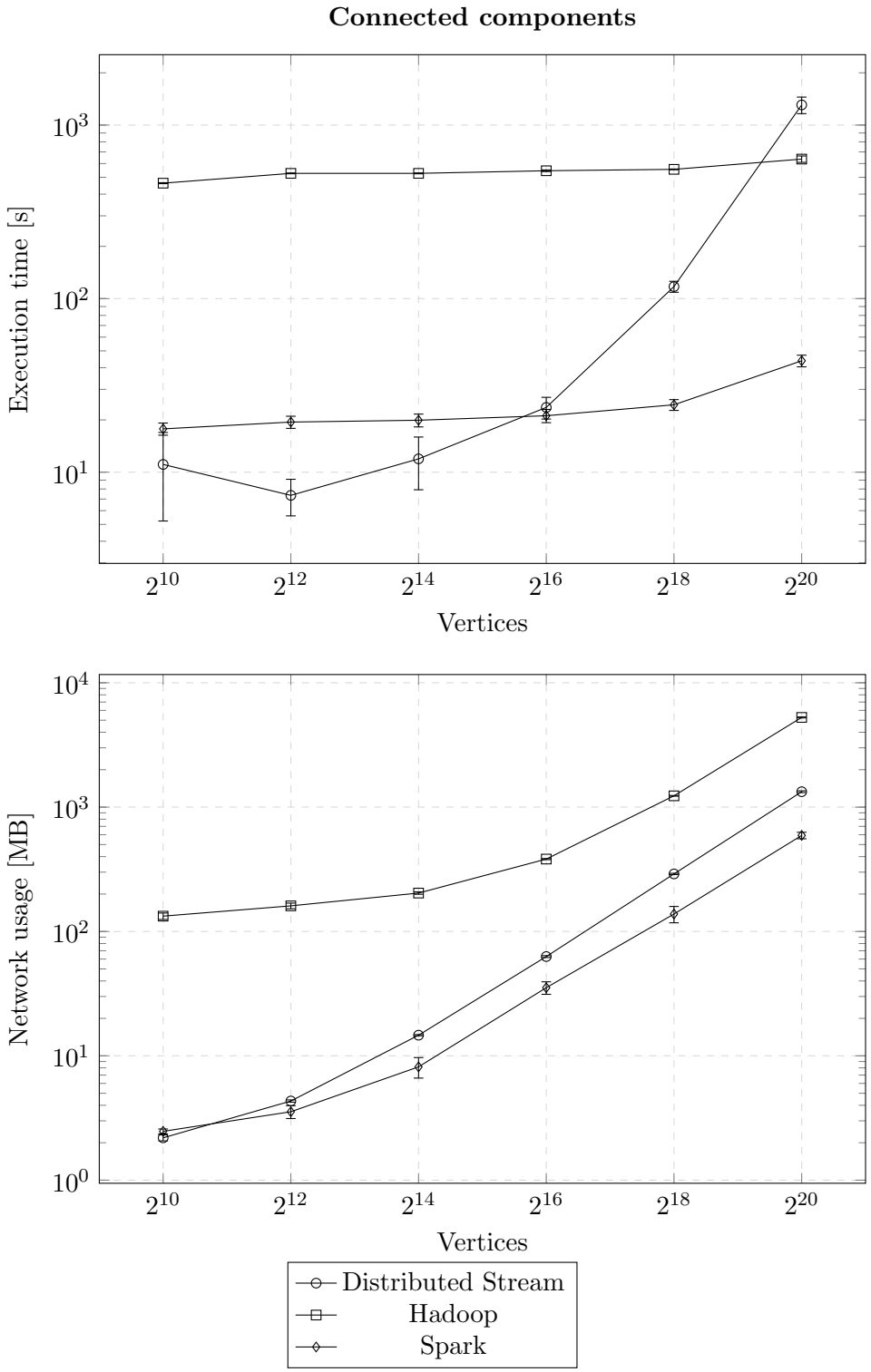


Figure A.14: Average execution time and network usage for the *Connected components* workload on cluster 2a (10 nodes).

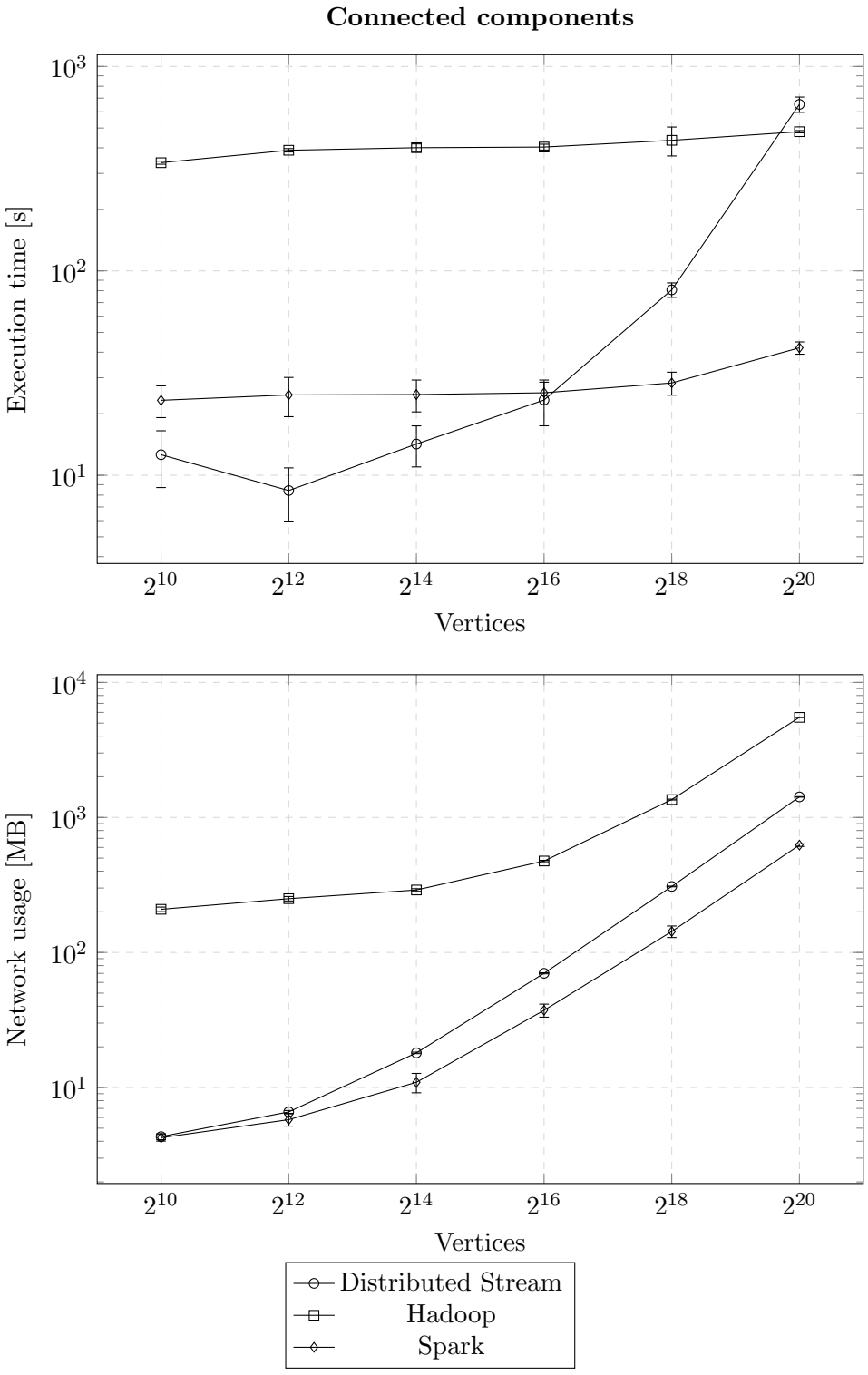


Figure A.15: Average execution time and network usage for the *Connected components* workload on cluster 2b (20 nodes).

Connected components	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	23.61	0.33	1	0	2266.15	5.26	100	0	18.20	0.47	0	0
2^{12}	25.37	0.29	4	0	2268.30	3.38	108	1	18.84	0.56	0	0
2^{14}	32.49	0.46	14	0	2594.66	5.80	160	1	20.83	0.49	1	0
2^{16}	62.08	0.72	62	0	2664.90	3.48	341	4	25.32	0.60	3	0
2^{18}	288.81	7.80	301	2	3047.09	6.77	1335	14	39.29	1.12	14	0

Table A.13: Average execution time and network usage for the *Connected components* workload on cluster 1 (6 nodes).

Connected components	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	11.08	5.85	2	0	462.50	3.02	133	8	17.76	1.43	2	0
2^{12}	7.35	1.75	4	0	526.77	2.80	161	8	19.44	1.56	4	0
2^{14}	11.93	4.01	15	0	526.78	3.04	204	4	19.92	1.69	8	2
2^{16}	23.61	3.38	63	1	545.47	5.88	382	4	21.14	1.85	35	4
2^{18}	117.30	8.52	290	3	555.61	2.78	1233	8	24.44	1.75	138	21
2^{20}	1306.40	142.91	1330	22	636.64	26.90	5265	48	43.83	3.38	592	35

Table A.14: Average execution time and network usage for the *Connected components* workload on cluster 2a (10 nodes).

Connected components	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	12.61	3.90	4	0	338.44	5.78	209	8	23.27	4.12	4	0
2^{12}	8.42	2.45	7	0	389.32	7.44	250	10	24.73	5.38	6	1
2^{14}	14.22	3.23	18	0	400.64	17.60	290	7	24.83	4.41	11	2
2^{16}	23.35	5.89	70	1	403.56	13.25	476	6	25.33	3.21	37	4
2^{18}	80.78	6.55	308	2	435.40	70.18	1356	11	28.30	3.65	143	14
2^{20}	652.53	55.93	1420	6	479.74	6.18	5513	23	42.03	2.90	624	14

Table A.15: Average execution time and network usage for the *Connected components* workload on cluster 2b (20 nodes).

A.6 PageRank

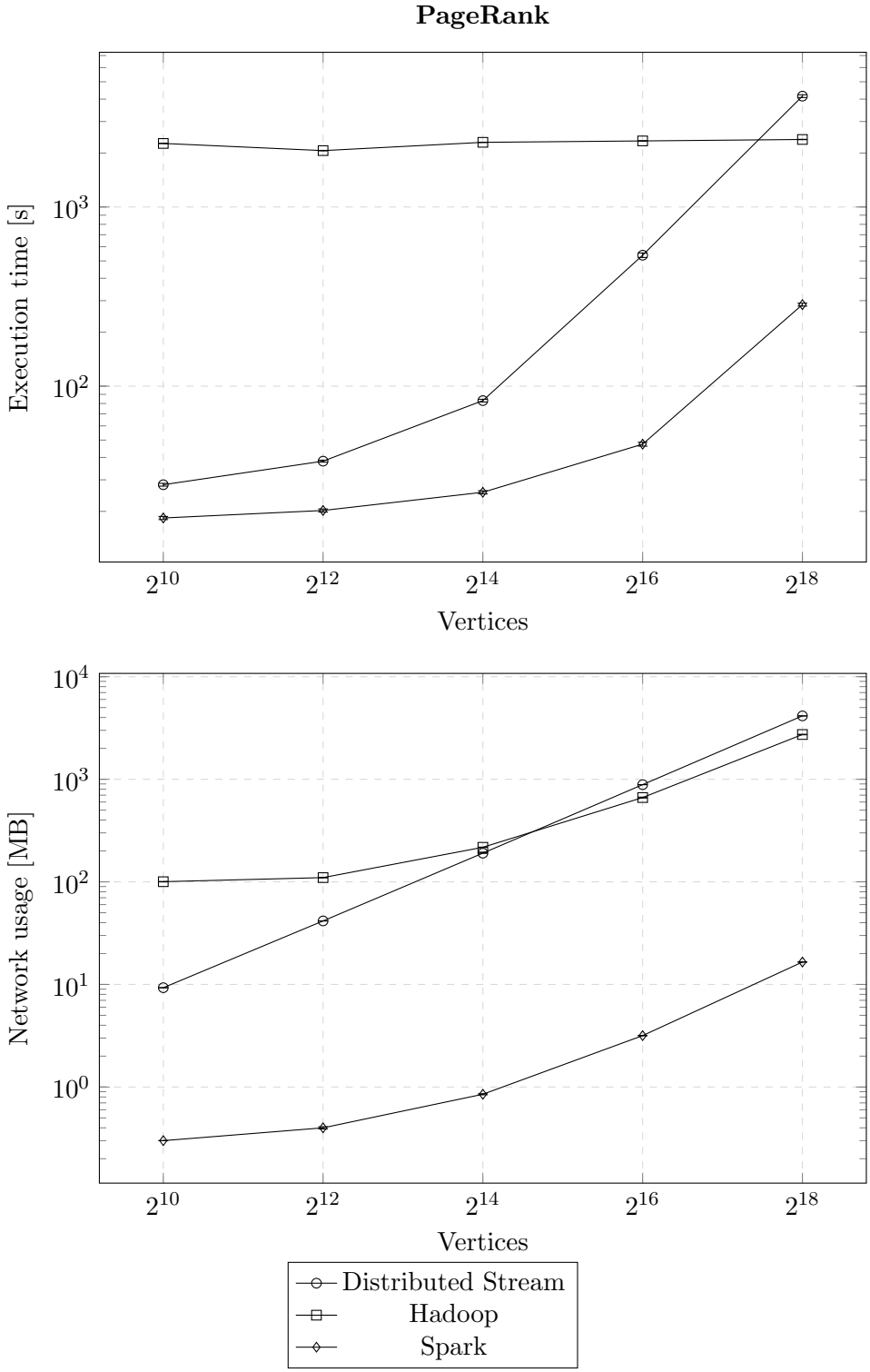


Figure A.16: Average execution time and network usage for the *PageRank* workload on cluster 1 (6 nodes).

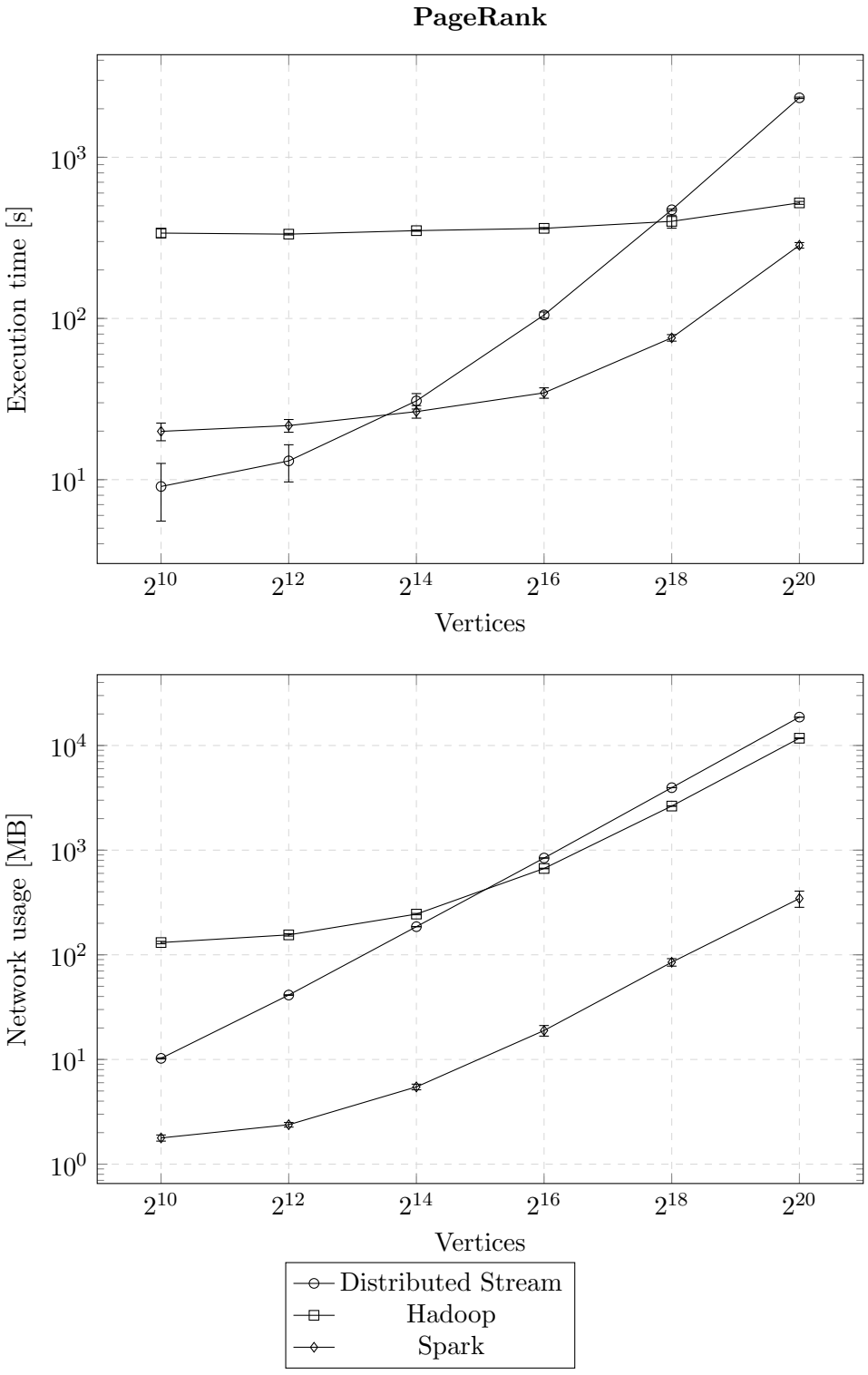


Figure A.17: Average execution time and network usage for the *PageRank* workload on cluster 2a (10 nodes).

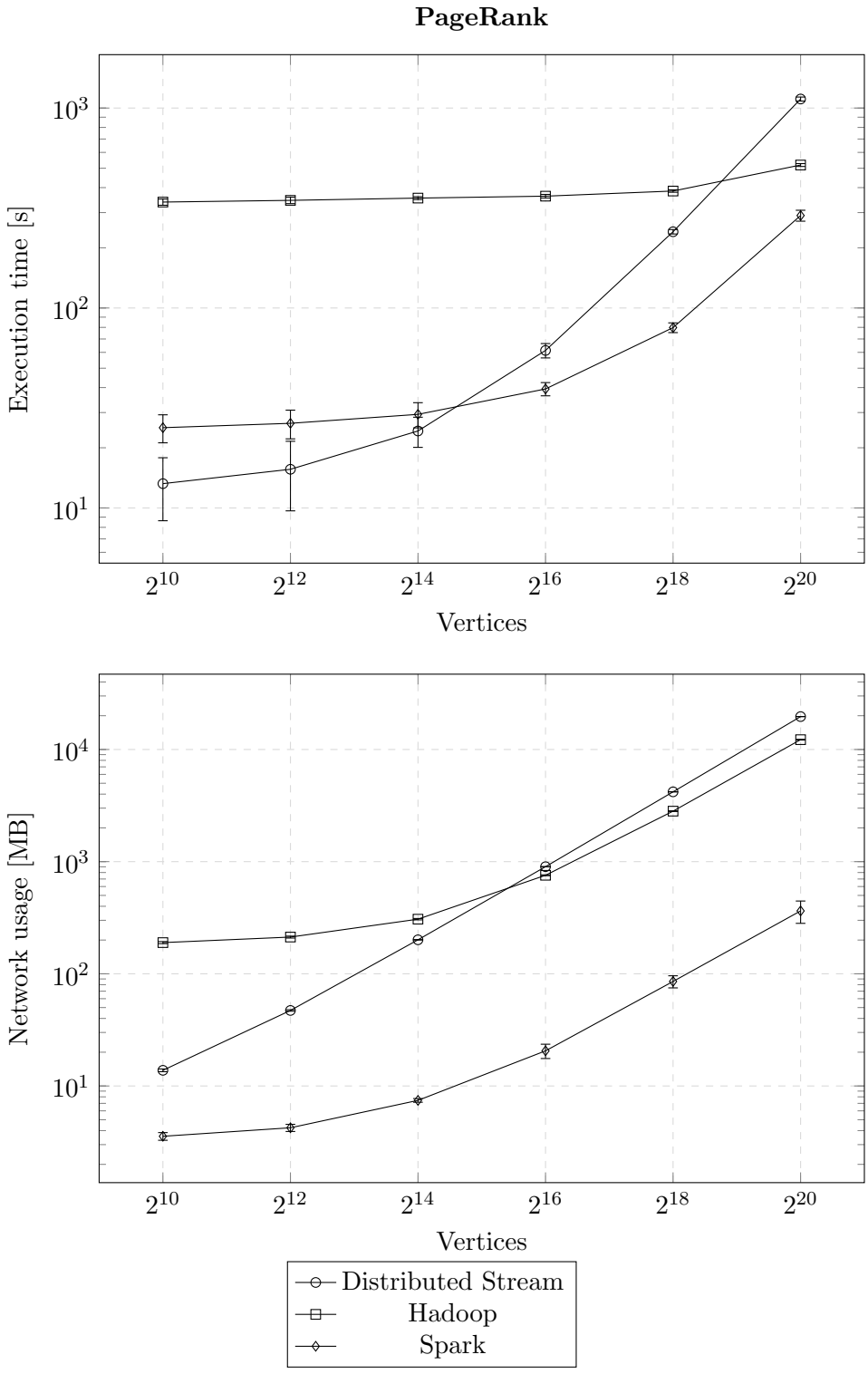


Figure A.18: Average execution time and network usage for the *PageRank* workload on cluster 2b (20 nodes).

PageRank	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	28.17	0.50	9	0	2263.28	5.26	100	0	18.37	0.37	0	0
2^{12}	38.13	0.48	42	0	2063.80	4.32	110	0	20.23	0.41	0	0
2^{14}	83.03	1.45	190	0	2295.45	3.77	217	1	25.56	0.53	1	0
2^{16}	538.09	15.45	886	0	2337.15	4.62	663	7	47.44	1.20	3	0
2^{18}	4156.69	72.46	4147	1	2381.45	5.07	2730	16	285.18	5.47	17	0

Table A.16: Average execution time and network usage for the *PageRank* workload on cluster 1 (6 nodes).

PageRank	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	9.07	3.54	10	0	338.60	21.33	131	4	19.94	2.50	2	0
2^{12}	13.07	3.39	41	0	333.49	3.80	155	4	21.65	1.96	2	0
2^{14}	30.81	3.39	186	0	350.66	3.08	245	4	26.43	2.36	5	0
2^{16}	105.08	4.66	841	1	362.36	4.57	668	4	34.56	2.56	19	2
2^{18}	472.29	6.12	3953	4	400.41	36.85	2638	16	75.86	3.53	85	7
2^{20}	2339.02	23.13	18664	16	521.32	10.80	11735	98	284.38	11.37	345	61

Table A.17: Average execution time and network usage for the *PageRank* workload on cluster 2a (10 nodes).

PageRank	Distributed Stream				Hadoop				Spark			
Input (vertices)	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.	Time (s)	Std. dev.	Network (MB)	Std. dev.
2^{10}	13.23	4.60	14	0	338.66	12.10	190	4	25.20	4.03	4	0
2^{12}	15.61	5.94	47	1	345.41	12.77	213	4	26.50	4.34	4	0
2^{14}	24.26	4.17	201	1	354.53	5.90	307	4	29.38	4.23	7	0
2^{16}	61.34	5.06	902	2	362.51	7.46	756	7	39.37	2.99	21	3
2^{18}	240.65	6.11	4192	5	384.58	5.44	2820	19	79.80	4.46	85	11
2^{20}	1109.54	25.61	19651	12	518.82	8.59	12234	90	289.98	18.22	364	82

Table A.18: Average execution time and network usage for the *PageRank* workload on cluster 2b (20 nodes).

Appendix B

Code listings

This appendix contains implementations of the workloads used in evaluating Distributed Streams for easy reference. Source code for the Distributed Stream implementation (chapter 6) and single-node experiments (chapter 4) can be found online at the follwing URL:

`https://github.com/RTSYork/DistributedStream`

B.1 Evaluation workloads

B.1.1 *Grep*

```
import dstream.*;
import dstream.util.*;

import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;
import java.util.stream.*;

public class HDFSGrep
{
  public static void main(String[] argv)
  {
    if (argv.length < 3)
    {
      System.err.println("Usage:␣Grep␣REGEXP␣INDIR␣OUTDIR");
      System.exit(1);
    }
    HDFSStringCollection lines = new HDFSStringCollection(argv[1]);
    HDFSStringCollectionWriter result = new HDFSStringCollectionWriter(argv[2]);
    grep(lines, result, argv[0]);
    result.finish();
  }

  static void grep(DistributedCollection<String> lines,
    DistributedCollection<String> result, String re)
  {
    Pattern match = Pattern.compile(re);
    // Find lines containing the regular expression
```

```
    DistributedMap<String, Long> localFound = lines
      .parallelStream()
      .flatMap(line ->
      {
        Matcher m = match.matcher(line);
        List<String> li = new ArrayList<>();
        while (m.find())
          li.add(m.group());
        return li.stream();
      })
      // Group local occurrences and count
      .localCollect(Collectors.groupingBy(s -> s, DistributedHashMap<String, Long>::new,
        Collectors.counting()));
    // Merge with all occurrences
    DistributedMap<String, Long> totalFound = localFound
      .entrySet()
      .parallelStream()
      .map(e -> new AbstractMap.SimpleEntry<>(e))
      .distribute(e -> e.getKey().hashCode())
      .localCollect(Collectors.groupingBy(e -> e.getKey(),
        DistributedHashMap<String, Long>::new,
        Collectors.summingLong(e -> (long) e.getValue())));
    localFound = null;
    // Store results
    totalFound
      .entrySet()
      .stream()
      .localForEach(e -> { result.add(e.getValue() + "\t" + e.getKey()); });
  }
}
```

B.1.2 *Sort*

```
import dstream.*;
import dstream.util.*;

import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.regex.*;
import java.util.stream.*;

public class HDFSSort
{
  public static void main(String[] argv)
  {
    if (argv.length < 2)
    {
      System.err.println("Usage:␣HDFSSort␣INDIR␣OUTDIR");
      System.exit(1);
    }
    HDFSStringCollection lines = new HDFSStringCollection(argv[0]);
    HDFSStringCollectionWriter result = new HDFSStringCollectionWriter(argv[1]);
```

```
    sort(lines, result);
    result.finish();
  }

  static void sort(DistributedCollection<String> lines, DistributedCollection<String> result)
  {
    // Sort and store results
    lines
      .stream()
      .sorted()
      .localForEach(i -> { result.add(i); });
  }
}
```

B.1.3 *Word-count*

```
import dstream.*;
import dstream.util.*;

import java.util.*;
import java.util.regex.*;
import java.util.stream.*;

public class HDFSWordCount
{
  public static void main(String[] argv)
  {
    if (argv.length < 2)
    {
      System.err.println("Usage:␣HDFSWordCount␣INDIR␣OUTDIR");
      System.exit(1);
    }
    HDFSStringCollection lines = new HDFSStringCollection(argv[0]);
    HDFSStringCollectionWriter result = new HDFSStringCollectionWriter(argv[1]);
    wordcount(lines, result);
    result.finish();
  }

  static void wordcount(DistributedCollection<String> lines,
    DistributedCollection<String> result)
  {
    Pattern delim = Pattern.compile("\\s+");
    // Get local word counts
    Map<String, Long> localCount = lines
      .parallelStream()
      .flatMap(line -> Stream.of(delim.split(line)))
      .localCollect(Collectors.groupingByConcurrent(w -> w, Collectors.counting()));
    DistributedMap<String, Long> localCountDistrib = DistributedMap.wrap(localCount);
    // Shuffle
    Map<String, Long> totalCount = localCountDistrib
      .entrySet()
      .parallelStream()
      .map(e -> new AbstractMap.SimpleEntry<>(e)) // Map.Entry is not serialisable
```

```
      .distribute(e -> e.getKey().hashCode())
      .localCollect(Collectors.groupingByConcurrent(e -> e.getKey(),
        Collectors.summingLong(e -> (long) e.getValue())));
    DistributedMap<String, Long> totalCountDistrib = DistributedMap.wrap(totalCount);
    // Get total counts
    totalCountDistrib
      .entrySet()
      .stream()
      .localForEach(e -> { result.add("(" + e.getKey() + "," + e.getValue() + ")"); });
  }
}
```

B.1.4 *Bayes*

```
import dstream.*;
import dstream.util.*;

import java.io.*;
import java.util.*;
import java.util.regex.*;
import java.util.stream.*;

import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

// Naive Bayes trainer and classifier
public class HDFSBayes
{
  private static void usage()
  {
    System.err.println("Usage:");
    System.err.println("␣␣␣␣HDFSBayes␣train␣INFILE␣TRAINFILE");
    System.err.println("␣␣␣␣HDFSBayes␣classify␣INFILE␣OUTFILE␣TRAINFILE");
    System.exit(1);
  }

  public static void main(String[] argv)
  {
    if (argv.length < 3)
      usage();
    HDFSStringCollection lines = new HDFSStringCollection(argv[1]);
    if (argv[0].equals("train"))
      train(lines, argv[2]);
    else if (argv[0].equals("classify"))
    {
      if (argv.length < 4)
        usage();
      HDFSStringCollectionWriter out = new HDFSStringCollectionWriter(argv[2]);
      classify(lines, out, argv[3]);
      out.finish();
    }
```

```
  else
    usage();
}

static void train(DistributedCollection<String> lines, String outFile)
{
  FileSystem fs = null;
  Configuration conf = new Configuration();
  conf.addResource(new Path(System.getenv("HADOOP_HOME") + "/conf/core-site.xml"));
  try
  {
    fs = FileSystem.get(conf);
  }
  catch (IOException e)
  {
    e.printStackTrace();
    System.exit(1);
  }
  ComputeNode root = ComputeGroup.getCluster().get(0);
  ComputeGroup rootGroup = new ComputeGroup(root);
  Pattern delim = Pattern.compile("\\s+");
  Map<String, P<HashMap<String, Integer>, Integer>> models_rk = lines
    .parallelStream()
    // Train every line
    .map(line -> {
      HashMap<String, Integer> wcount = new HashMap<>();
      String[] tokens = delim.split(line);
      String category = tokens[0];
      for (int i = 0; i < tokens.length; i++)
        wcount.put(tokens[i], 1);
      return new P<>(category, new P<>(wcount, 1));
    })
    // Merge all classifiers
    .distribute(rootGroup)
    .localCollect(Collectors.groupingBy(p -> p.k,
      Collector.of(() -> new P<>(new HashMap<>(), 0), (p, a) -> {
      Map<String, Integer> wcount = p.k;
      for (String word: a.v.k.keySet())
        wcount.put(word, wcount.getOrDefault(word, 0) + 1);
      p.v += a.v.v;
    }, (p1, p2) ->
    {
      Map<String, Integer> wcount = p1.k;
      for (String word: p2.k.keySet())
        wcount.put(word, wcount.getOrDefault(word, 0) + 1);
      p1.v += p2.v;
      return p1;
    })));
  // Calculate probabilities
  if (ComputeNode.getSelf() == root)
  {
    Map<String, HashMap<String, Double>> models_p = models_rk
      .entrySet()
      .parallelStream()
```

```
      .map(p -> {
        HashMap<String, Double> prob = new HashMap<>();
        Map<String, Integer> wcount = p.getValue().getKey();
        double ccount = (double) p.getValue().getValue();
        for (String word: wcount.keySet())
          prob.put(word, (double) wcount.getOrDefault(word, 0) / ccount);
        return new P<>(p.getKey(), prob);
      })
      .collect(Collectors.toMap(P::getKey, P::getValue));

    try
    {
      ObjectOutputStream out = new ObjectOutputStream(fs.create(new Path(outFile), false));
      out.writeObject(models_p);
      out.close();
    }
    catch (IOException e)
    {
      e.printStackTrace();
      System.exit(1);
    }
  }
}

static void classify(DistributedCollection<String> lines,
  DistributedCollection<String> out, String trainFile)
{
  FileSystem fs = null;
  Configuration conf = new Configuration();
  conf.addResource(new Path(System.getenv("HADOOP_HOME") + "/conf/core-site.xml"));
  try
  {
    fs = FileSystem.get(conf);
  }
  catch (IOException e)
  {
    e.printStackTrace();
    System.exit(1);
  }
  Pattern delim = Pattern.compile("\\s+");
  // Read models into memory
  Map<String, Map<String, Double>> models;
  try
  {
    ObjectInputStream in = new ObjectInputStream(fs.open(new Path(trainFile)));
    models = (Map<String, Map<String, Double>>) in.readObject();
    in.close();
  }
  catch (IOException | ClassNotFoundException e)
  {
    e.printStackTrace();
    System.exit(1);
    return;
  }
```

```
    lines
      .parallelStream()
      .map(line -> {
        // Get words in line with no duplicates
        Set<String> words = Arrays.stream(delim.split(line)).collect(Collectors.toSet());
        double maxp = -1.0;
        String result = "default";
        // Iterate through each model and compute most probable model
        for (Map.Entry<String, Map<String, Double>> m: models.entrySet())
        {
          double p = 1.0;
          Map<String, Double> model = m.getValue();
          for (String word: words)
            p *= model.getOrDefault(word, 0.0001);
          if (p > maxp)
          {
            maxp = p;
            result = m.getKey();
          }
        }
        return result + "␣" + line;
      })
      .localForEach(line -> {
        out.add(line);
      });
  }

  // Serialisable key-value pair
  private static class P<T extends Serializable, U extends Serializable>
    implements Serializable
  {
    public T k; // key
    public U v; // value

    public P(T k, U v)
    {
      this.k = k;
      this.v = v;
    }

    public T getKey()
    {
      return k;
    }

    public U getValue()
    {
      return v;
    }
  }
}
```

B.1.5 *Connected components*

```
import dstream.*;
import dstream.util.*;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;
import java.util.stream.*;

public class HDFSConnComponents
{
  public static void main(String[] argv)
  {
    if (argv.length < 2)
    {
      System.err.println("Usage:␣HDFSConnComponents␣INFILE␣OUTDIR");
      System.exit(1);
    }
    HDFSStringCollection lines = new HDFSStringCollection(argv[0]);
    HDFSStringCollectionWriter outVertices =
      new HDFSStringCollectionWriter(argv[1] + "/vertices");
    HDFSStringCollectionWriter outSummary =
      new HDFSStringCollectionWriter(argv[1] + "/summary");
    cc(lines, outVertices, outSummary, 0);
  }

  static Queue<HashSet<Long>> localMerge(Queue<HashSet<Long>> q)
  {
    boolean changed;
    do
    {
      Queue<HashSet<Long>> q2 = new ConcurrentLinkedQueue<>();
      changed = false;
      while (!q.isEmpty())
      {
        HashSet<Long> s1 = q.remove();
        if (s1.isEmpty())
          continue;
        for (HashSet<Long> s2: q)
        {
          if (!Collections.disjoint(s1, s2))
          {
            s1.addAll(s2);
            s2.clear();
            changed = true;
          }
        }
        q2.add(s1);
      }
      q = q2;
    }
    while (changed);
```

```
  return q;
}

static void cc(DistributedCollection<String> lines, DistributedCollection<String> result,
  DistributedCollection<String> summary, int iterations)
{
  Pattern delim = Pattern.compile("\\t");
  // Load edges
  Queue<HashSet<Long>> localComponents = new ConcurrentLinkedQueue<>();
  lines
    .parallelStream()
    .flatMap(line -> {
      String[] ints = delim.split(line);
      if (ints.length != 2)
      {
        System.err.println("Invalid␣line␣format");
        System.exit(1);
      }
      long v1 = Long.parseLong(ints[0]);
      long v2 = Long.parseLong(ints[1]);
      if (v1 < v2)
        return Stream.of(new Edge(v1, v2));
      if (v1 > v2)
        return Stream.of(new Edge(v2, v1));
      return null;
    })
    .distribute(e -> (int) e.v1)
    .localForEach(e ->
    {
      HashSet<Long> s = new HashSet<>();
      s.add(e.v1);
      s.add(e.v2);
      localComponents.add(s);
    });
  // Merge local components
  Queue<HashSet<Long>> localMergedComponents = localMerge(localComponents);
  DistributedCollection<HashSet<Long>> components =
    DistributedCollection.wrap(localMergedComponents);
  // Merge all components: send to first node
  ComputeNode node0 = ComputeGroup.getCluster().get(0);
  boolean isNode0 = node0.isSelf();
  components
    .parallelStream()
    .filter(s -> !isNode0) // Don't send components in node0
    .distribute(node0)
    .localForEach(s -> { localMergedComponents.add(s); });
  if (!isNode0)
    localMergedComponents.clear();
  // Merge local components again
  DistributedCollection<HashSet<Long>> allComponents =
    DistributedCollection.wrap(localMerge(localMergedComponents));
  // Output number of components
  long count = allComponents
    .parallelStream()
```

```
      .count();
    System.out.println(ComputeNode.getSelf().getName() + ": There are " +
      count + " components");
    // Output vertices and their component IDs
    // Output summary (number of vertices in each component; ie. number of
    // vertices with same component ID)
    allComponents
      .stream()
      .localForEach(s ->
      {
        List<Long> sorted = new ArrayList<>(s);
        Collections.sort(sorted);
        long root = sorted.get(0);
        for (Long i: sorted)
          result.add(i + "\t" + root);
        summary.add(root + "\t" + s.size());
      });
  }
}

// Represents a directed edge from v1 to v2
class Edge implements Serializable
{
  public long v1, v2; // Vertices: v1 -> v2

  public Edge(long v1, long v2)
  {
    this.v1 = v1;
    this.v2 = v2;
  }
}
```

B.1.6 *PageRank*

```
import dstream.*;
import dstream.util.*;

import java.io.*;
import java.util.*;
import java.util.regex.*;
import java.util.stream.*;

public class HDFSPageRank
{
  public static void main(String[] argv)
  {
    if (argv.length != 3)
    {
      System.err.println("Usage: HDFSPageRank INFILE OUTDIR ITERATIONS");
      System.exit(1);
    }
    HDFSStringCollection lines = new HDFSStringCollection(argv[0]);
    HDFSStringCollectionWriter out = new HDFSStringCollectionWriter(argv[1]);
```

```
  int iter = Integer.parseInt(argv[2]);
  pagerank(lines, out, iter);
  out.finish();
}

static void pagerank(DistributedCollection<String> lines,
  DistributedCollection<String> out, int iterations)
{
  Pattern delim = Pattern.compile("\\t");
  // Directed edges: (from, (to1, to2, to3))
  System.out.println("Loading␣links");
  Map<String, List<String>> localLinks = lines
    .parallelStream()
    .map(s -> {
      String[] parts = delim.split(s);
      return new P<String, String>(parts[0], parts[1]);
    })
    .distribute(p -> p.getKey().hashCode())
    .localDistinct()
    .localCollect(Collectors.groupingByConcurrent(p -> p.getKey(),
      Collectors.mapping(p -> p.getValue(), Collectors.toList())));
  DistributedMap<String, List<String>> links = DistributedMap.wrap(localLinks);

  System.out.println(links.size() + "␣links␣loaded.");
  // Initialise ranks to 1.0
  Map<String, Double> localRanks = links
    .keySet()
    .parallelStream()
    .localCollect(Collectors.toConcurrentMap(s -> s, s -> 1.0,
      (a, b) -> a));
  DistributedMap<String, Double> ranks = DistributedMap.wrap(localRanks);

  for (int it = 0; it < iterations; it++)
  {
    System.out.println("Iteration␣" + (it + 1));
    final DistributedMap<String, Double> curRanks = ranks;
    // Calculate contribution to destinations
    Map<String, Double> localContribs = links
      .entrySet()
      .parallelStream()
      .flatMap(e -> {
        double size = (double) e.getValue().size();
        return e
          .getValue()
          .stream()
          .map(url -> new P<String, Double>(url,
            curRanks.getOrDefault(e.getKey(), 0.0) / size));
      })
      .distribute(p -> p.getKey().hashCode())
      // Add contributions for each destination together
      .localCollect(Collectors.toConcurrentMap(
        p -> p.getKey(), p -> p.getValue(),
        (v1, v2) -> v1 + v2));
    DistributedMap<String, Double> contribs = DistributedMap.wrap(localContribs);
```

```
      // Adjust ranks
      localRanks = contribs
        .entrySet()
        .parallelStream()
        .map(p -> new P<>(p.getKey(), p.getValue() * 0.85 + 0.15))
        .localCollect(Collectors.toConcurrentMap(P::getKey, P::getValue, (v1, v2) -> v1));
      ranks = DistributedMap.wrap(localRanks);
    }
  }

  // Represents a key-value pair
  private static class P<T extends Serializable, U extends Serializable>
    implements Serializable
  {
    public T k; // key
    public U v; // value

    public P(T k, U v)
    {
      this.k = k;
      this.v = v;
    }

    public T getKey()
    {
      return k;
    }

    public U getValue()
    {
      return v;
    }
  }
}
```

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[2] A. M. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 167–176, 1999.

[3] Amazon Web Services, Inc. Amazon Simple Storage Service (S3). `https://aws.amazon.com/s3/`, accessed 2016/06/30.

[4] Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org/`, accessed 2015/12/09.

[5] Apache Software Foundation. Apache HBase. `https://hbase.apache.org/`, accessed 2016/06/30.

[6] Apache Software Foundation. Apache Hive^{TM}. `https://hive.apache.org/`, accessed 2015/08/21.

[7] Apache Software Foundation. Apache Mahout: Scalable machine learning and data mining. `http://mahout.apache.org/`, accessed 2015/08/21.

[8] Apache Software Foundation. Apache Spark – Lightning-Fast Cluster Computing. `http://spark.incubator.apache.org/`, accessed 2013/10/03.

[9] Apache Software Foundation. Examples — Apache Spark. `http://spark.apache.org/examples.html`, accessed 2015/12/09.

[10] Apache Software Foundation. GraphX. `https://spark.apache.org/graphx/`, accessed 2016/05/01.

[11] Apache Software Foundation. HDFS Architecture Guide. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`, accessed 2016/01/04.

[12] Apache Software Foundation. MapReduce Tutorial. `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`, accessed 2015/12/09.

[13] Apache Software Foundation. Spark Programming Guide. `http://spark.apache.org/docs/latest/programming-guide.html`, accessed 2015/08/21.

[14] Apache Software Foundation. The Apache Cassandra Project. `https://cassandra.apache.org/`, accessed 2016/06/30.

[15] Nathan Backman, Rodrigo Fonseca, and Uğur Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 1:1–1:5, New York, NY, USA, 2012. ACM.

[16] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[17] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[18] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[19] Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1999.

[20] Carnegie Mellon University. Pegasus: Peta-Scale Graph Mining System. `http://www.cs.cmu.edu/~pegasus/`, accessed 2016/05/01.

[21] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[22] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[23] Intel Corporation. Intel Cilk Plus Language Specification. `https://www.cilkplus.org/sites/default/files/open_specifications/cilk_plus_language_specification_0_9.pdf`, accessed 2015/11/24.

[24] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 987–998, New York, NY, USA, 2010. ACM.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] J. Diaz, C. Munoz-Caro, and A. Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, August 2012.

[27] École Polytechnique Fédérale de Lausanne. The Scala Programming Language. `http://www.scala-lang.org/`, accessed 2015/12/01.

[28] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[29] Iman Elghandour and Ashraf Aboulnaga. Restore: reusing results of mapreduce jobs. *Proceedings of the VLDB Endowment*, 5(6):586–597, 2012.

[30] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 4(9):575–585, June 2011.

[31] T. Epperly, A. Prantl, and B. Chamberlain. Composite Parallelism: Creating Interoperability between PGAS Languages, HPCS Languages and Message Passing Libraries. 2011.

[32] Ericsson AB. Erlang Programming Language. URL: `http://www.erlang.org/`, accessed 2016/01/04.

[33] Graham E. Fagg and Jack J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, August 2004.

[34] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[35] Matthew Finnegan. Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic. `http://www.computerworlduk.com/news/data/boeing-787s-create-half-terabyte-of-data-per-flight-says-virgin-atlantic-3433595/`, accessed 2015/11/24.

[36] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[37] A. Grama, A. Gupta, G. Karypis, and V Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, 2003.

[38] W. Gropp and E. Lusk. Fault Tolerance in MPI Programs. *Special issue of the Journal High Performance Computing Applications (IJHPCA)*, 18:363–372, 2002.

[39] W. D. Gropp and R. Thakur. Issues in Developing a Thread-Safe MPI Implementation. In *PVM/MPI*, pages 12–21, 2006.

[40] haskell.org. Haskell Language. `https://www.haskell.org`, accessed 2015/12/01.

[41] IBM Big Data & Analytics Hub. Infographic: The Four V's of Big Data. URL: `http://www.ibmbigdatahub.com/infographic/four-vs-big-data`, accessed 2015/11/24.

[42] iMatix Corporation. Distributed Messaging - zeromq. URL: http://zeromq.org/, accessed 2016/01/04.

[43] Jacek Radajewski and Douglas Eadline. Beowulf HOWTO. URL: http://ibiblio.org/pub/Linux/docs/HOWTO/archive/Beowulf-HOWTO.html, accessed 2016/01/04.

[44] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[45] Alekh Jindal, Jorge Quiané-Ruiz, and Samuel Madden. Cartilage: Adding flexibility to the hadoop skeleton. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1057–1060, New York, NY, USA, 2013. ACM.

[46] Alan Kaminsky. Parallel Java 2 Library. http://www.cs.rit.edu/~ark/pj2.shtml, accessed 2014/08/07.

[47] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

[48] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[49] LBNL / UC Berkeley. GASNet Communication System. URL: http://gasnet.lbl.gov/, accessed 2016/01/04.

[50] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006.

[51] M. Levy, R. Oshana, D. Stewart, and M. Domeika. *Multicore Programming Practices*. The Multicore Association, 2013.

[52] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[53] Sam Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, May 2012.

[54] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, June 2011.

[55] Bernard Marr. Why only one of the 5 Vs of big data really matters. URL: http://www.ibmbigdatahub.com/blog/why-only-one-5-vs-big-data-really-matters, accessed 2015/11/24.

[56] Martin Grotzke. EsotericSoftware/kryo: Java serialization and cloning: fast, efficient, automatic. https://github.com/EsotericSoftware/kryo, accessed 2016/06/10.

[57] Nathan Marz. Storm – Distributed and fault-tolerant realtime computation. http://storm-project.net/, accessed 2013/10/03.

[58] Eileen McNulty. Understanding Big Data: The Seven V's. http://dataconomy.com/seven-vs-big-data/, accessed 2015/11/24.

[59] Cristbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329, 2 2014.

[60] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[61] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[62] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., 1996.

[63] OpenSFS and EOFS. Lustre. http://lustre.org/, accessed 2016/06/10.

[64] Oracle Corporation. ForkJoinPool (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html, accessed 2015/08/21.

[65] Oracle Corporation. java.util.stream (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, accessed 2016/03/14.

[66] Oracle Corporation. JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107, accessed 2013/09/05.

[67] Oracle Corporation. JSR 335: Lambda Expressions for the Java(TM) Programming Language. https://jcp.org/en/jsr/detail?id=335, accessed 2015/06/01.

[68] Oracle Corporation. Stream (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, accessed 2014/11/17.

[69] Fengfeng Pan, Yinliang Yue, Jin Xiong, and Daxiang Hao. I/o characterization of big data workloads in data centers. In Jianfeng Zhan, Rui Han, and Chuliang Weng, editors, *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, volume 8807 of *Lecture Notes in Computer Science*, pages 85–97. Springer International Publishing, 2014.

[70] Marcel Proust. Remembrance of Things Past. `http://alarecherchedutempsperdu.com/text.html`, accessed 2013/09/04.

[71] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[72] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., first edition, 2007.

[73] Rich Hickey. Clojure. `clojure.org`, accessed 2015/12/01.

[74] Harvey Richardson. High performance fortran: history, overview and current developments. Technical report, 1.4 TMC-261, Thinking Machines Corporation, 1996.

[75] Nathan Rutman. Map/Reduce on Lustre. URL: `http://www.xyratex.com/sites/default/files/Xyratex_white_paper_MapReduce_1-4.pdf`, accessed 2016/01/04, 2011.

[76] V. A. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. A Brief Introduction To X10 (For the High Performance Programmer). URL: `http://x10.sourceforge.net/documentation/intro/latest/html/`, accessed 2013/06/18, 2012.

[77] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. Technical report, Toronto, Canada, June 2010.

[78] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The Hadoop Distributed Filesystem: Balancing Portability and Performance, 2010.

[79] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core HPC systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009.

[80] Sun Microsystems, Inc. Java SE 6 Performance White Paper. `http://www.oracle.com/technetwork/java/6-performance-137236.html`, accessed 2015/08/21.

[81] Sayantan Sur, Hao Wang, Jian Huang, Xiangyong Ouyang, and Dhabaleswar K Panda. Can high-performance interconnects benefit hadoop distributed file system? In *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO*, 2010.

[82] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[83] Typesafe Inc. Akka. URL: `http://akka.io/`, accessed 2016/01/04.

[84] Xabier Cid Vidal and Ramon Cid Manzano. Taking a closer look at LHC. `http://www.lhc-closer.es/1/3/12/0`, accessed 2014/05/05.

[85] Abhinav Vishnu, Huub Van Dam, Wibe de Jong, Pavan Balaji, and Shuaiwen Song. Fault-tolerant communication runtime support for data-centric programming models. In *2010 International Conference on High Performance Computing, HiPC 2010, Dona Paula, Goa, India, December 19-22, 2010*, pages 1–9. IEEE Computer Society, 2010.

[86] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 119–128, Oct 2014.

[87] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, K. Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499, Feb 2014.