2019-07-01

A Shared-Memory Coupled Architecture to Leverage Big Data Frameworks in Prototyping and In-Situ Analytics for Data Intensive Scientific Workflows

Alexander Michael Lemon
Brigham Young University

A Shared-Memory Coupled Architecture to Leverage Big Data Frameworks in Prototyping and In-Situ Analytics for Data Intensive Scientific Workflows

Alexander Michael Lemon

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

ABSTRACT

A Shared-Memory Coupled Architecture to Leverage Big Data Frameworks in Prototyping and In-Situ Analytics for Data Intensive Scientific Workflows

Alexander Michael Lemon
Department of Computer Science, BYU
Master of Science

There is a pressing need for creative new data analysis methods which can sift through scientific simulation data and produce meaningful results. The types of analyses and the amount of data handled by current methods are still quite restricted, and new methods could provide scientists with a large productivity boost. New methods could be simple to develop in big data processing systems such as Apache Spark, which is designed to process many input files in parallel while treating them logically as one large dataset. This distributed model, combined with the large number of analysis libraries created for the platform, makes Spark ideal for processing simulation output.

Unfortunately, the filesystem becomes a major bottleneck in any workflow that uses Spark in such a fashion. Faster transports are not intrinsically supported by Spark, and its interface almost denies the possibility of maintainable third-party extensions. By leveraging the semantics of Scala and Spark's recent scheduler upgrades, we force co-location of Spark executors with simulation processes and enable fast local inter-process communication through shared memory. This provides a path for bulk data transfer into the Java Virtual Machine, removing the current Spark ingestion bottleneck.

Besides showing that our system makes this transfer feasible, we also demonstrate a proof-of-concept system integrating traditional HPC codes with bleeding-edge analytics libraries. This provides scientists with guidance on how to apply our libraries to gain a new and powerful tool for developing new analysis techniques in large scientific simulation pipelines.

Keywords: Apache Spark, Data-Intensive Science, High-Performance Computing, In-Situ Analytics, Parameter Sweep, State-Space Pruning

ACKNOWLEDGEMENTS

A special thanks is due to the staff and attending physicians at Utah Valley Hospital, who saved my life multiple times during the process of writing this thesis. I could also not have done this without the support of my family. To everyone else who helped in the creation of this document: thank you for listening.

Contents

Tables

Figures

1 Introduction

Modern high-performance computing (HPC) produces a lot of data. This is especially true of simulation codes. Some "... are producing petabytes and even exabytes of data," according to the data-intensive science team at Argonne National Laboratory [1]. The massive data produced by these simulations is useless unless it can be efficiently analyzed. Unfortunately, even storing data at this scale is difficult, and analyzing it is even more challenging. There is a pressing need for creative new data analysis methods which can sift through this mass of data and produce meaningful results.

For a specific example, consider the simulation in nuclear reactor design. Nuclear scientists must simulate various reactor parts in a wide variety of conditions. In one study, with a restricted case with few parameters, scientists still had to run 6561 simulations for a total of about 18 hours of CPU time [2]. While scientists have managed to get by, the types of analyses and the amount of data handled by current methods are still very restricted. This has led to complex systems such as JobPruner[3], which integrates with job schedulers to heuristically skip job submissions which are likely unproductive.

Systems like JobPruner are difficult to modify and tune, and they are perhaps more limited than most systems by the main bottleneck of coupled workflows: the filesystem. Generally, many simulations have to run and store data in files; these files are then analyzed serially, after which further simulations may be run, and the original simulations must be rerun if any problems were detected during analysis. As a result, workflows take

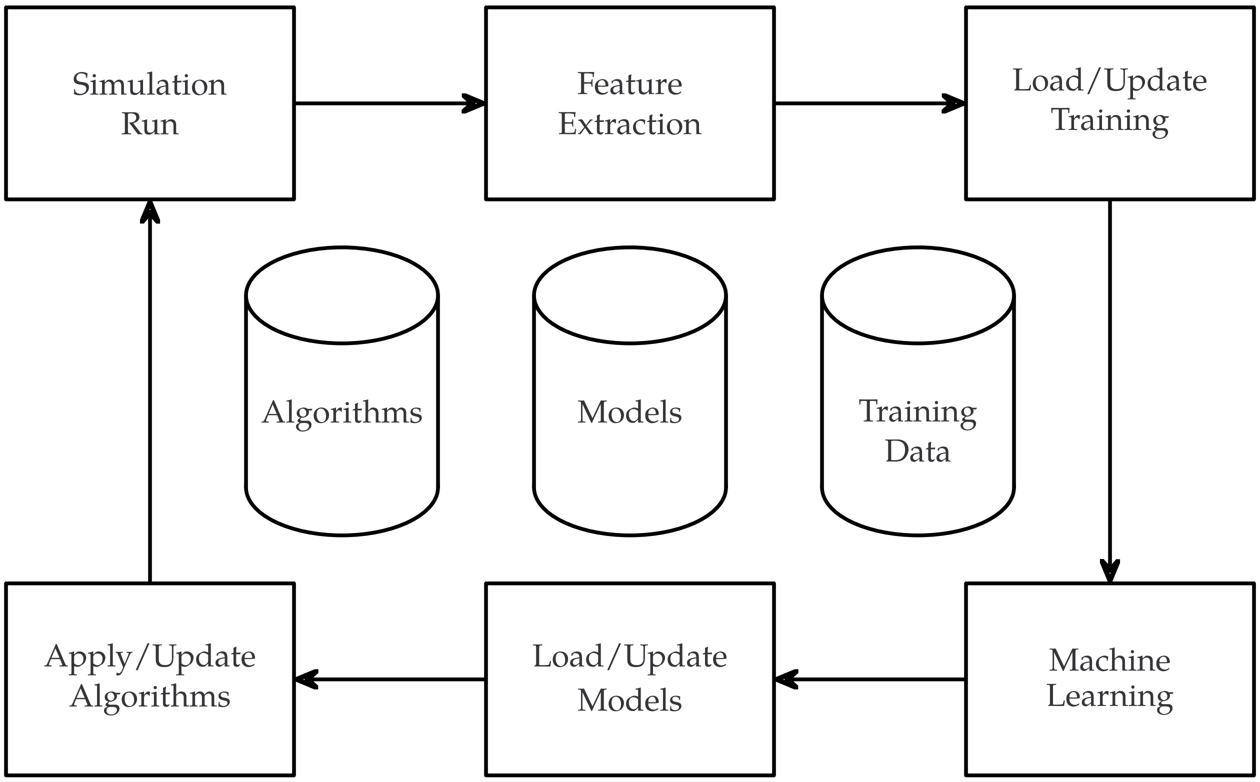

Figure 1.1 Traditional Coupled Workflows

the form shown in **Figure 1.1**, where standard runs combined with feedback mechanisms can result in as many as six major filesystem bottlenecks in a single application.

These limitations seem artificial given current technology; there are now systems designed to do general statistical analyses on large amounts of data in parallel. Unfortunately, these systems were not developed to integrate with HPC, but rather with the large amounts of data produced by business processes, internet-of-things devices, and a wide variety of other real-world data. One especially popular framework is Apache Spark [4], which aims to be a one-stop-shop for analysis of warehoused and streaming data. It provides distributed machine learning algorithms, SQL queries, and most of the basic tools required for doing any sort of distributed analysis. In theory, Spark could reduce the filesystem hits in **Figure 1.1** from six to two simply because it subsumes all the pieces other than the simulation that produces data.

Spark is designed to process many input files in parallel while treating them logically as one large in-memory distributed dataset, so the intermediate I/O is gone, leaving only the interface with simulations. This makes Spark ideal for processing simulation output, except for one thing: Big data systems, including Spark, generally rely on distributed filesystems to handle the most complex parts of distributed processing and fault tolerance, so these systems almost always read from and write to disk on every job they run. Network filesystem I/O is one of the biggest overheads in running on a supercomputer, which means that HPC applications try to avoid it, so Spark's reliance on the filesystem has thus far prevented Spark from being applied directly in HPC environments. To enable efficient coupling, processes need to send data to Spark through fast transports such as shared memory rather than filesystems.

If Spark could ingest data from anything other than filesystems, databases, and low-throughput network interfaces, HPC might already be using it. It is not trivial to change Spark's capabilities, though, because its limitations are mostly caused by the Java Virtual Machine (JVM). The Spark core libraries have to do amazingly complicated workarounds to deal with bottlenecks such as 32-bit array indices. Most attempts to circumvent Spark's sanctioned interfaces fail because they run into the same JVM problems that Spark has managed to mitigate. Spark uses a complex custom memory management scheme which provides, to end users, the illusion of effortless management of large address spaces. Unfortunately, this management scheme is not designed for interoperability, so taking advantage of it is best done by exploiting Scala language features such as lazy singleton instantiation; this can yield mostly stable results without requiring a rewrite of large parts of Spark.

1.1 Thesis Statement

Any solution for connecting data-intensive HPC to Apache Spark must solve two problems: it must ensure co-location of the two applications, and it must use inter-process communication (IPC) transports which work with existing Spark APIs. By ensuring and facilitating co-location of Spark executors with HPC processes, we can enable the use of local IPC methods such as shared memory. By using a ring buffer of shared memory, we can efficiently transport bulk data into the JVM, removing the current Spark ingestion bottleneck. Removing this bottleneck will create a significant speedup in Spark-coupled simulation pipelines. This speedup will facilitate using Spark (and its ecosystem) to accelerate the development of new analysis methods for scientific simulation.

2 Related Work

Truly related works are scarce in this field. We examine the closest existing applications, HPC workflow systems, to show the opportunity for using Spark. Then we show existing projects which integrate with Spark in some fashion.

				Interoperability Cost	
System	**Scheduling Flexibility**	**Analytic Functionality**	**State**	**Native**	**Spark**
Ignite	Low	Low	Production	High	Low
H2O.ai	Low	High	Production	High	Low
OpenCPU	High	High	Research	Medium	Medium
SnappyData	High	High	Production	High	Low
Dask	High	Medium	Alpha	Low	NA
JuliaDB	Medium	Medium	Alpha	Low	NA
pbdR	High	Medium	Research	Low	NA

Table 2.1 Feature Matrix of Current Systems

The important features are summarized in **Table 2.1** to demonstrate the lack of a system which is sufficiently usable in all the areas needed for our application.

The trade-offs apparent in our table come because Spark is difficult to integrate with other systems, so most projects directly using Spark are based entirely on the JVM platform and on other projects in the Spark ecosystem. Unfortunately, the JVM does not mesh well with supercomputing applications, and introduces all kinds of memory-management issues and overheads.

We group projects by how they manage control flow between Spark and user code and whether Spark is augmenting the application or vice-versa. In addition to projects from the Spark ecosystem, we also examine three possible alternatives to Spark which all have better potential for efficient coupling. We also examine interoperability tools which could help solve the problem of coupling native and JVM code.

2.1 Workflow Systems

We looked first at existing workflow systems to confirm there was a Spark-shaped gap in current tooling. While workflow systems abound, many are highly domain-specific. One general-purpose system stands out as representative, in that it captures both the strengths and weaknesses of most existing HPC workflow systems.

Dakota [5] is a general system for providing iterative analyses to HPC simulations. It provides both loose- and tightly-coupled modes. Loose coupling in Dakota is based on pairs of input/output files for each program, and is representative of typical simulation-analysis workflows. Tightly-coupled mode embeds the simulation into Dakota's process, which allows very efficient data exchange between the simulation and analysis halves of the workflow. The problem with tightly-coupled mode is it does require modification of the simulation program, although a wrapper based on Filesystem in Userspace (FUSE) could probably be introduced for simulations which cannot be modified. In general, Dakota seems useful, but only for situations where its built-in analyses are sufficient. Implementing new and experimental machine learning and other analysis methods would likely be difficult, as supporting these is not a goal of Dakota.

The main issue with existing workflow frameworks is a lack of flexibility. Thus they can't compete with Spark's ecosystem for ease-of-prototyping. If an HPC framework decided to prioritize prototyping and managed to keep its performance and robustness intact, it could easily remove the need for Spark. Converting an existing framework to do such a task would be a massive undertaking, as would developing a new framework with feature parity.

2.2 Spark-Centric

Some projects achieve tight integration with Spark by creating custom Spark executors which encapsulate instances of their program.

Ignite [6] uses this method to enable tight integration of Spark Resilient Distributed Datasets (RDDs) with the Ignite data grid. Unfortunately, using this tightly coupled mode precludes the use of an existing Ignite cluster, and is thus not useful for integrating with processes external to Spark.

H2O.ai [7] integrates with Spark through a sub-project named Sparkling Water. This project enables the use of H2O's Machine Learning APIs; this is good for avoiding duplication of effort in designing distributed machine learning algorithms, but as with Ignite, provides no way to connect to external processes.

OpenCPU [8] provides a custom Spark executor which can offload various statistical computations to a cluster of R processes. This project actually presents a real possibility for coupling Spark to external systems which themselves embed R. Unfortunately, this library currently focuses on the case where data exists in the Spark process and must be shipped to the external cluster. It doesn't seem to provide for situations in which data exists on the external processes and needs to be sampled by the Spark process. The OpenCPU infrastructure might prove useful, but it is not yet being utilized in a way that solves the problem at hand.

2.3 Subsume Spark

The ideal method of communicating with Spark would be to have it run in-process with the data-producing code. There is currently only one system which embeds Spark into itself in this manner.

SnappyData [9] is essentially a peer-to-peer data grid with Spark executors embedded into the data serving nodes. This provides excellent data locality, as all data transfer between Spark and the grid (which is also on the JVM) is in-process. To achieve this tight coupling, SnappyData has to maintain its own Spark scheduling components; it has actually forked the Spark code base and embedded it in its repository. SnappyData requires a large amount of maintenance to stay in sync with Spark's main release branch, which is undesirable. The biggest flaw for our purposes is that SnappyData hasn't solved any general Spark integration problems (ie. the embedding of Spark into arbitrary applications); it merely reduces them to the problem of how to integrate with SnappyData, which is nontrivial to do in an efficient way, especially from a non-JVM process.

2.4 Language Interop

Babel [10] was named "The world's most rapid communication among many programming languages in a single application" by the R&D100. This award was given in 2006, and the framework was showcased at Supercomputing 2007 and used for some time, but it seems to have been inactive for about five or six years at present. In any case, Babel was mostly concerned with single-process applications which happened to use modules written in different languages. It provided some support for remote method invocation, but was certainly not intended for use as a general framework for data movement between co-located but mostly independent applications. If resurrected, Babel might contribute insights on high performance interfaces between native code and the JVM, but would not be a useful solution to the Spark problem by itself.

Weld [11] is an intermediate representation (IR) designed for representing SQL operators and similar functions in a language-agnostic fashion. Weld is brand new, and the developers have not published an efficient framework for code generation, so Weld specifically might not be useful. However, Weld has demonstrated that generating native code from Spark is possible and that it increases performance. This capability could be applied to function shipping. That is, Weld does not care where the data on which it operates is located when the code is being generated. This means that the IR could be generated and shipped across a network to where the data resides, and the result shipped back. By design, Weld could, in fact, be generated partially by a library on the JVM, then further modified by operations from a native library before being executed on arbitrary data. Using Weld in this fashion could at least provide an optimization in the form of query push-down, but could also form the basis for a complete redesign of the Spark processing model.

2.5 Embeddable Alternatives

Because embedding Spark is so difficult, completely switching analysis systems could be a reasonable alternative. We investigated three potential replacements for Spark which in theory would be far easier to embed.

2.5.1 Dask

Our first alternative, Dask[12], was one we investigated early on in our research. We discarded it because (a.) it was nearly brand new, (b.) it did not officially support machine learning, and (c.) it had breaking bugs.

Later in our project we had cause to re-evaluate and found that Dask had greatly improved and had better documentation and support for machine learning. Unfortunately, Dask still manifested bugs when we attempted to test our workloads with it.

If Dask is developed further with an emphasis on correctness and it can become stable, then we believe it could be an excellent alternative to Spark for building coupled simulation-analysis workflows. In fact, because many Spark users prefer the PySpark variant, they would prefer Dask to Spark simply because Dask is a Python framework and complex Spark functionality requires the use of Scala.

2.5.2 JuliaDB

JuliaDB[13] is, like Dask, not mature yet. In fact, we only discovered it when we were doing our second evaluation of Dask. JuliaDB is developed with commercial support and has the potential to be a reasonable alternative to Dask or Spark, but it is currently far too immature to be useful.

In addition to its own youth, JuliaDB has the problem of being written in Julia, which is also immature and facing design challenges. If Julia could overcome certain development issues, especially the way it handles library dependencies and environments, then Julia and JuliaDB would be excellent candidates for embedding in compiled programs. In theory, the language's built-in interoperability with C exceeds that of Python and R, so JuliaDB could even be the best of our three alternatives. Whether this potential will ever be realized is doubtful at this point.

Julia recently released a version 1.0 which didn't address most of the problems we faced with the language. This leaves little hope that Julia and JuliaDB will ever be suitable for this type of project.

2.5.3 Programming with Big Data in R

Abbreviated pbdR[14], this framework has a traditional supercomputing background and is developed by academics and national lab scientists. Its main contribution is an idiomatic R interface to distributed and high-performance linear algebra libraries. While pbdR integrates well with other HPC applications, it does not currently provide a large library of useful machine learning functions, as Dask and JuliaDB do in theory and as Spark does in practice. If developers were to create such a library, this framework might be the most functional of the alternatives we tested. It builds mostly on proven HPC technologies whose bugs were mostly resolved long ago.

2.5.4 Summary

We tested the most promising frameworks for three popular scientific and statistical computing languages, and found each of them lacking. Each of them, if truly completed, would be a strong contender and probably replace Spark for projects like ours. Unfortunately, none of the frameworks is mature and as a result, important functionality, correctness, or both were absent from each one. So while there are promising alternatives, the promises will be fulfilled too far in the future for the alternatives to be useful for our work. Perhaps the best feature of Spark is that, while it does have flaws, *it exists now*, its interface contains a large amount of functionality, and its implementation mostly works as promised. Spark won't always be the best framework, but it's currently the only framework that isn't vaporware.

3 Implementation

Our framework is agnostic concerning specific parallel libraries used in simulation codes, so it would be equally correct to substitute arbitrary single-program multiple-data (SPMD) processes in most places where we write MPI (Message Passing Interface). MPI, though, is occasionally more convenient to use in discussion, and we also used MPI for our own testing, so we hereafter refer to simulation processes as MPI processes.

Our solution addresses a number of outstanding issues for coupling Spark and traditional HPC codes. First, we ensure that Spark workers and MPI processes are co-located one-to-one, as shown in the left block of **Figure 3.1**. We also smoothly coordinate the launch and shutdown of both frameworks to facilitate co-location. Second, we provide a simple, low-overhead node-local data transport between MPI and Spark workers. This transport is the data channel (along with its connected components) shown in **Figure 3.1**. The transport's protocol requires a metadata channel, which is drawn above it in the figure. Third, we offer a remote procedure call (RPC) interface through which MPI processes can request arbitrary analytics on the datasets resulting from our second contribution via the Spark driver. As shown in **Figure 3.1**, our RPC system is based on the hypertext transfer protocol (HTTP).

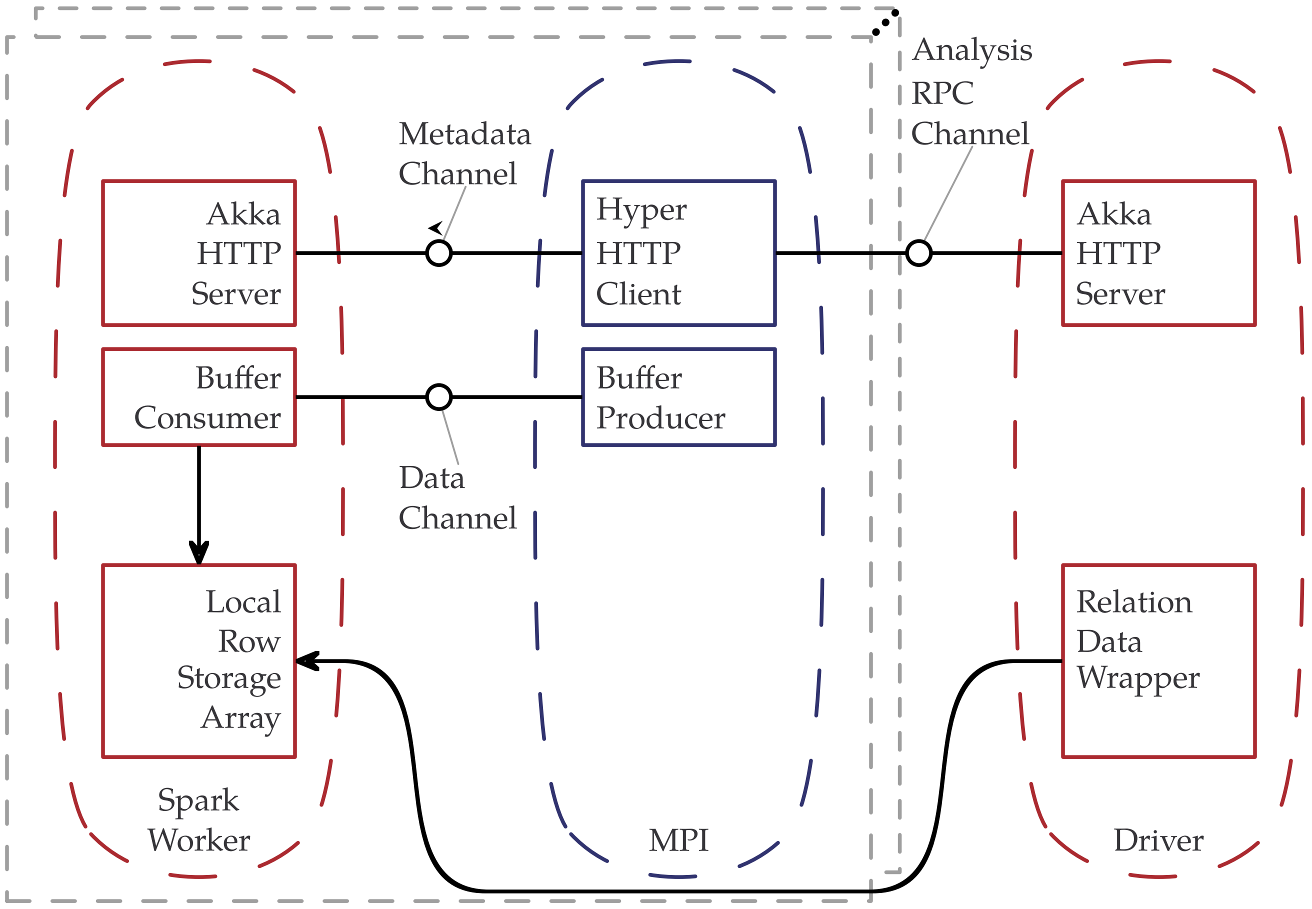

Figure 3.1 Coupling Detail

All of our framework's runtime components are conveniently packaged as Java Archives (JARs) and shared libraries which spare uninterested users the burden of dealing with implementation details. Potential users only need three skills:

1. The ability to write Scala methods conforming to an interface
2. An average understanding of Apache Spark
3. The ability to inject a few lines of code (resembling standard C I/O routines) in an MPI program.

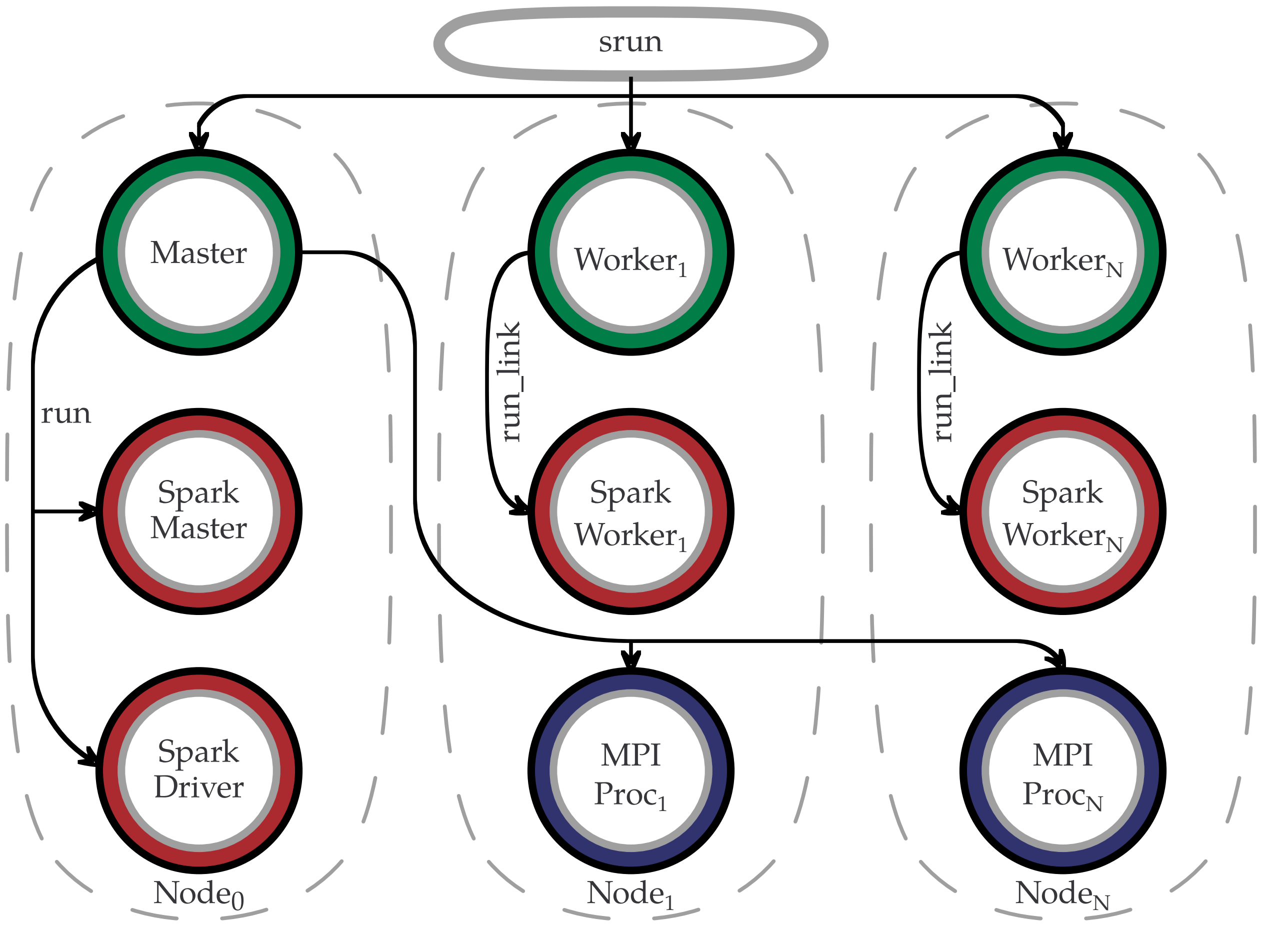

Figure 3.2 Colored processes implicitly communicate during startup. Arrows show connections between Slurm and the supervisor, and between the supervisor and its child processes.

Originally the only way to reliably co-locate anything with Spark was by using Apache Mesos as Spark's scheduler. With newer versions of Spark, this fortunately became unnecessary, but Spark by itself is difficult to manage. Specifically, Spark creates complexity with the handling of various log files, routing of non-file output from its numerous components, controlling startup, ensuring graceful shutdown, and the integration of all these functions into a job submitted via an HPC scheduler such as Slurm. To handle these issues we wrote a small supervisor framework which intercepts, for example, dying Spark jobs to increase relevance of the job exit status for accounting. The supervisor also separates logs for all the processes from both halves of the coupled application to prevent trashing the standard Slurm output collector. We use this supervisor to cleanly launch, monitor the profiling and benchmark applications we discuss in the following chapter.

In practice, the supervisor operates as shown in **Figure 3.2**. Processes from the same application are shown in different colors and share common label prefixes. In the figure, arrows denote explicit communication between groups, and each group also performs internal communications for initialization.

- The green nodes form a semi-connected Erlang actor system.
- The red nodes run Spark's internal RPC registration routines.
- The blue nodes perform an MPI_Init call during startup.

Note that while the actor system is supervising both Spark and an MPI program, Spark and MPI processes do not communicate during launch. Our protocol (detailed in the next section) requires that Spark be initialized before the MPI processes. This requirement ensures that simulations cannot attempt to send data to a service which is not yet listening.

3.2 Transport

Our transport method makes two major improvements over the traditional filesystem approach. First, it operates in-memory, avoiding any bottlenecks with local disk and with the cluster storage network. This feature also makes our transport indifferent to differences in cluster setup; eg. its performance is invariant to the presence or absence of local disks on compute nodes. Second, our transport avoids the serialization-deserialization (serde) overhead which plagues most existing frameworks and which can be just as expensive as the actual data transfer. Serde is often mandatory because arbitrary data models moving between arbitrary machines are rarely binary-compatible. Because all our transfer is between co-located processes, we can exploit raw data representations and avoid serde for cross-machine purposes. We also avoid any serde that might be used for language interoperability because all types in our data model are interoperable between Scala and C.

3.2.1 Buffer

Our transport method is based on shared memory. We access ***/dev/shm*** directly due to lack of full JVM support for shared memory as defined by the Portable Operating System Interface (POSIX). This places a Linux dependency on our implementation, which should be a non-issue in practice because a large proportion of HPC clusters run Linux. Several major non-Linux systems have the ability to provide a compatible replacement for ***/dev/shm***, and even without a Linux compatibility module, could support the necessary operations if a few lines of one Scala file were edited. The stronger dependencies our system has are on POSIX IPC; systems must support many of the standard constructs for our system to work correctly.

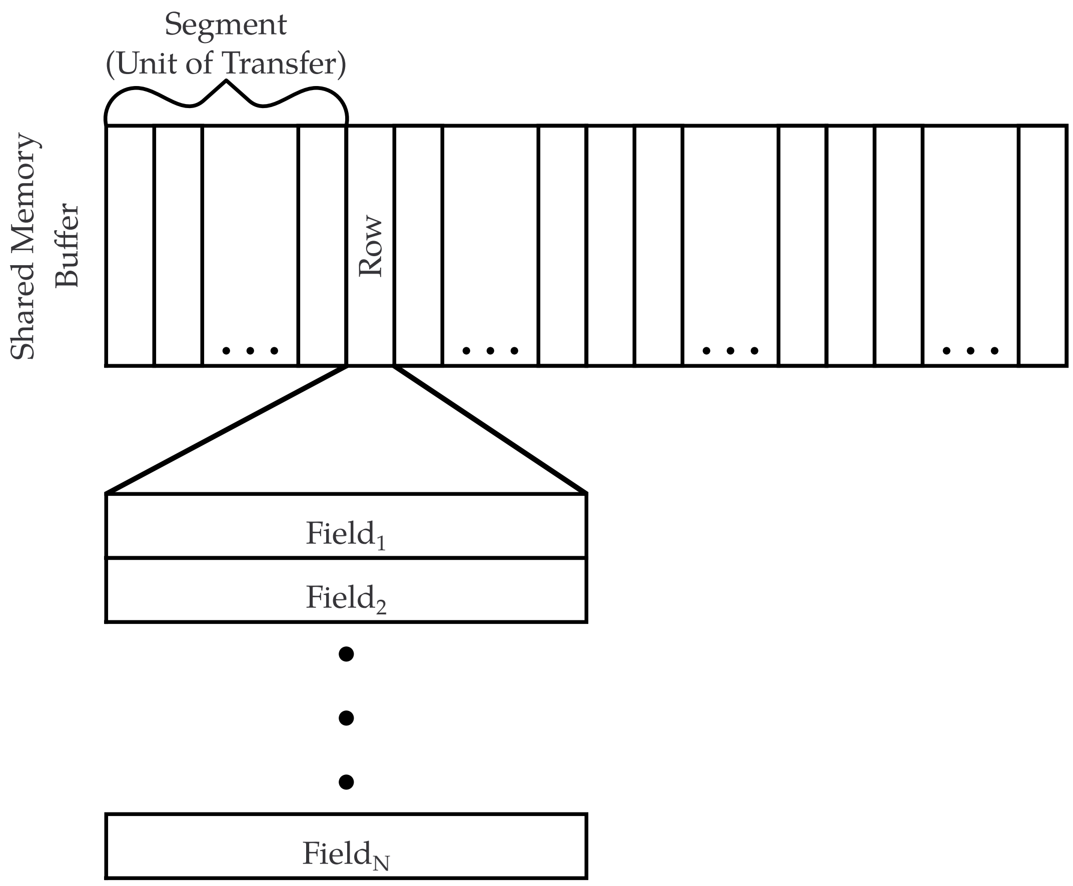

Figure 3.3 The "Ring" in "Ring Buffer" contains indexed segments comprised of rows. In a fully-threaded implementation, segments could be randomly accessable and lockable, but our single-producer-consumer paired version uses semaphores and a single modular counter instead. All segments are transferred only when full, except for the very last, which is allowed to be partially empty.

Our buffer is formed from a conservatively sized slab of memory-mapped shared memory (shown in **Figure 3.3**). The segments of this buffer are groups of rows. Transferring less than a certain amount of data in one segment is bad for performance, as it greatly increases the number of copy calls as well as the number of semaphore operations. The segment size should be set to minimize this under average conditions; calibrating it per-application or per-cluster-configuration is advisable. Since users rarely transfer multiples of the segment size, we allow the last segment to be partially empty when transferred.

3.2.2 Protocol

The actual transfer of data is a classic single-producer-consumer setup, with a semaphore for signaling a segment is ready and another for signaling a segment is empty. Both processes keep their own modular index for tracking which segment to operate on next. A third semaphore is used to signal the end of the transfer operations, so that one process can clean up the resources before continuing.

Because flaws in this protocol would break our entire framework, we took great care to design it correctly. We modeled the protocol in Promela; refer to **Figure 3.4** for the pieces. We cared about two main properties:

i. $\square\,(ring_0 = 0 \lor ring_0 = 1)$ (Mutual Exclusion)

ii. $(\Diamond\,(to\_transfer > 0)) \land \square\,(to\_transfer > 0 \implies \Diamond\,(to\_transfer = 0))$ (Completeness)

The properties use linear temporal logic, and for reference we provide the intuitive (though imprecise) definitions of the unary operators $\square$ and $\Diamond$.

- $\square$ This operator, often read *globally*, specifies that its operand (a formula) should hold in every state.

```
#define  SLOTS  4
#define  SLICES  12
byte ring[SLOTS];
byte sem_full;
byte sem_empty = SLOTS - 1;
byte to_transfer = SLICES;
```

Initial State

```
inline  sem_wait(sem) {
   atomic {
      (sem > 0);
      sem--;
   }
}
inline  sem_post(sem) {
   sem++;
}
```

Semaphores

```
active  proctype  mpi_process() {
   byte segment = 0;
   do
      ::  to_transfer > 0 -> {
         assert(ring[segment] == 0);
         ring[segment]++;
         segment = (segment + 1) %
SLOTS;
         sem_post(sem_full);
         sem_wait(sem_empty);
      }
      ::  else -> break
   od
}
```

Producer

```
active  proctype  spark_executor() {
   byte segment = 0;
   do
      ::  to_transfer > 0 -> {
         sem_wait(sem_full);
         assert(ring[segment] == 1);
         ring[segment]--;
         to_transfer--;
         segment = (segment + 1) %
SLOTS;
         sem_post(sem_empty);
      }
      ::  else -> break
   od
}
```

Consumer

Figure 3.4 Protocol Model

◊ Sometimes called *eventually*, this operator requires that its operand be true either in the current state or in some future state.

Mutual exclusion is a standard property in concurrent systems with shared resources. Completeness is our own property we wrote to check the end-to-end behavior of the protocol, and would fail if undesirable properties such as deadlock could occur. It ensures that when any data becomes available from the producer, that data is eventually consumed. Before moving on with our work, we used SPIN to ensure that our model satisfied both properties. A full state-space search for the corresponding never claims (with acceptance cycles) runs without errors for each property. By carefully implementing our verified protocol, we more easily unmasked subtle bugs which manifested during development and which, if not for our careful planning, would have led to many wrong debugging paths through unrelated code.

To enable our simple setup, both processes need to be on the same page concerning the schema and amount of data being transferred. Each process also needs to call initialization methods to work with the POSIX IPC objects. We satisfy both these requirements with a single HTTP exchange. When native processes have data to transfer, they send a message, containing the metadata for the upcoming transfer, to their local Spark workers. Our framework, embedded in Spark, has precisely one HTTP endpoint; this endpoint takes an incoming message and calls a receiving method configured with the message's metadata body.

3.2.3 Data Model

The second feature of our transport is the most influential in its design. Raw data is only useable if programmers haven't shaped it into complex structures.

Type	Typical Size (bits)
Char	8
Int	32
Long	64
Float	32
Double	64

Table 3.1 All data types are *signed* scalar values corresponding to those implemented in typical C compilers (Clang, GCC) when targeting x86.

We thus restrict our data model to a common subset usable across most language environments: records with signed scalar fields. For reference, the possible field types are given in **Table 3.1**; they correspond to the set of types mutually understood by Java and C. By design, this matches the standard Spark data model, which is in turn designed around a subset of SQL. The trick of our model restriction is that we use the *exact same* model everywhere in the coupled system, to minimize conversion costs.

While many low-level transport methods are byte-oriented, our row-oriented model motivated us to make our transport row-oriented as well. For users to transfer data to Spark, they simply need their data in the form of an array of structs (AoS). Not all user data fits this model well, but although we considered making the transfer library use format strings or schema files or some such structure to interpret and transform user data before transferring to Spark, we decided the associated complexity and the serde overhead would both be too high. In order to avoid serde, users have to transform their data somehow, as all raw binary methods are inherently constrained in similar ways. As a result, a number of existing codes whose data is not, in theory, a good fit for our system might already include suitable conversions for other I/O systems.

While we cannot do anything about codes whose data model is difficult to transform to AoS form, it is better that users of such codes know the cost of using Spark up front. It is easier to reason about conversion costs when the loops for flattening data are in user code, not hidden in a transport library. We do assist users as much as possible once they have a structure definition for the elements of an AoS; when they provide a C structure, we generate the Spark side of the interface for them.

This may sound trivial, but Scala's type system is often painful for new users. The type system cannot be avoided in our framework because we require certain compile-time information to ensure transferred data is interpreted correctly and to avoid spending any more time than necessary on reflection-based data conversion. To help users avoid this potential snare, we created a Clang-based code generator to complement the client libraries. By generating all the glue, we spare users from the internal details of the Scala data model for most common cases.

Users need only run the code generator and place the output in their application's source tree. As a result, users get the efficiency of compile-time optimizations with equal convenience to Spark's reflection-based schema inference.

3.3 Library

By nature, every piece of our library was actually two pieces. Our framework comes in two pieces: a C-compatible shared library to interface with simulation codes, and a JAR file to use with Spark.

3.3.1 Native

Our native library is designed for embedding in high-performance programs with C-compatibility layers; we wanted it to have a small footprint, few if any dependencies, and a simple interface. We also wanted it to be easy to write, verify, and maintain, and to have minimal chances of causing bugs due simply to implementation quirks. To this end, we designed the library to be stateless and almost entirely statically linked, and we wrote it in Rust.

We didn't start out with a stateless library. In fact, we first designed the library in the fashion typical of MPI, embedded language runtimes (Ada, Haskell, Lua, Scheme, etc), HDF5 and NetCDF, database drivers, or graphics libraries. In other words, we started down the traditional path followed by most C interface libraries. In C, libraries which deal with external state tend to have initializers which return handles which are then provided as arguments to other library calls. To avoid resource leaks, C programmers must always call finalizers on these handles so the external code can clean things up.

Initially, we set our library up the same way, but we noticed that we weren't actually storing state with our handle, so we refactored our library to get rid of the initialization and termination code. It turns out that the main pieces of our library which might theoretically be useful to keep around are the ones which tended to function best if we re-initialized them every time. So, instead of allocating a set of buffers and semaphores and maintaining them, we allocated one buffer and one set of semaphores when a transfer was initiated, then freed them immediately after the transfer completed. While this does come with a slight runtime cost, it decreases the chance of the program leaving garbage synchronization objects behind in the event of program failure. If the overhead had been severe, we would of course have found a way to use persistent state, but we did not encounter such problems in practice, and our simplified approach helped solve some very real and persistent synchronization debugging issues that were holding up development. Our stateless interface was also smaller than the stateful version. It had three methods: one to transfer data to executors, one to request analysis results, and one to free the resources associated with the analysis results.

Note that our library is C-compatible but is written in Rust because Rust provided useful development features that are missing from the C ecosystem.

- Debugging was greatly simplified by writing in Rust instead of C, thanks to built-in memory protection and useful backtraces.

- Rust was also useful for its package ecosystem, which has multiple well-maintained

HTTP libraries of varying complexity. Had we written in C or C++, we would have been hard-pressed to find even one HTTP library suitable for our purposes.

- The Rust build tool defaults to static linking, so the only runtime dependencies our library has are standard C libraries, which would already be required by most potential user code (the exceptions would likely be Fortran programs, and Fortran-C interoperability solutions would probably need the C libraries themselves). In addition to simplifying dependency management, static linking enables more efficient job startup on clusters. Static linking isn't mandatory, of course, but having it by default is a useful bonus of using Rust.

3.3.2 Scala

As we mentioned earlier, we generate code for Spark data model creation and loading. These parts of the Spark library are written as templates which are expanded for every struct provided by a user. The resulting code is responsible for storing, loading and casting data into the correct form for use in DataFrame-centric codes. No actual generated code is necessary to compile; empty stubs could be substituted. Our auto-generated file thus only impacts runtime functionality.

The rest of our library internals are split between executor code and driver code. Executors have services provided by Akka HTTP; each executor has an actor listening for metadata submissions. The driver also has an Akka HTTP-based service which listens for analysis requests and provides a simple form of RPC.

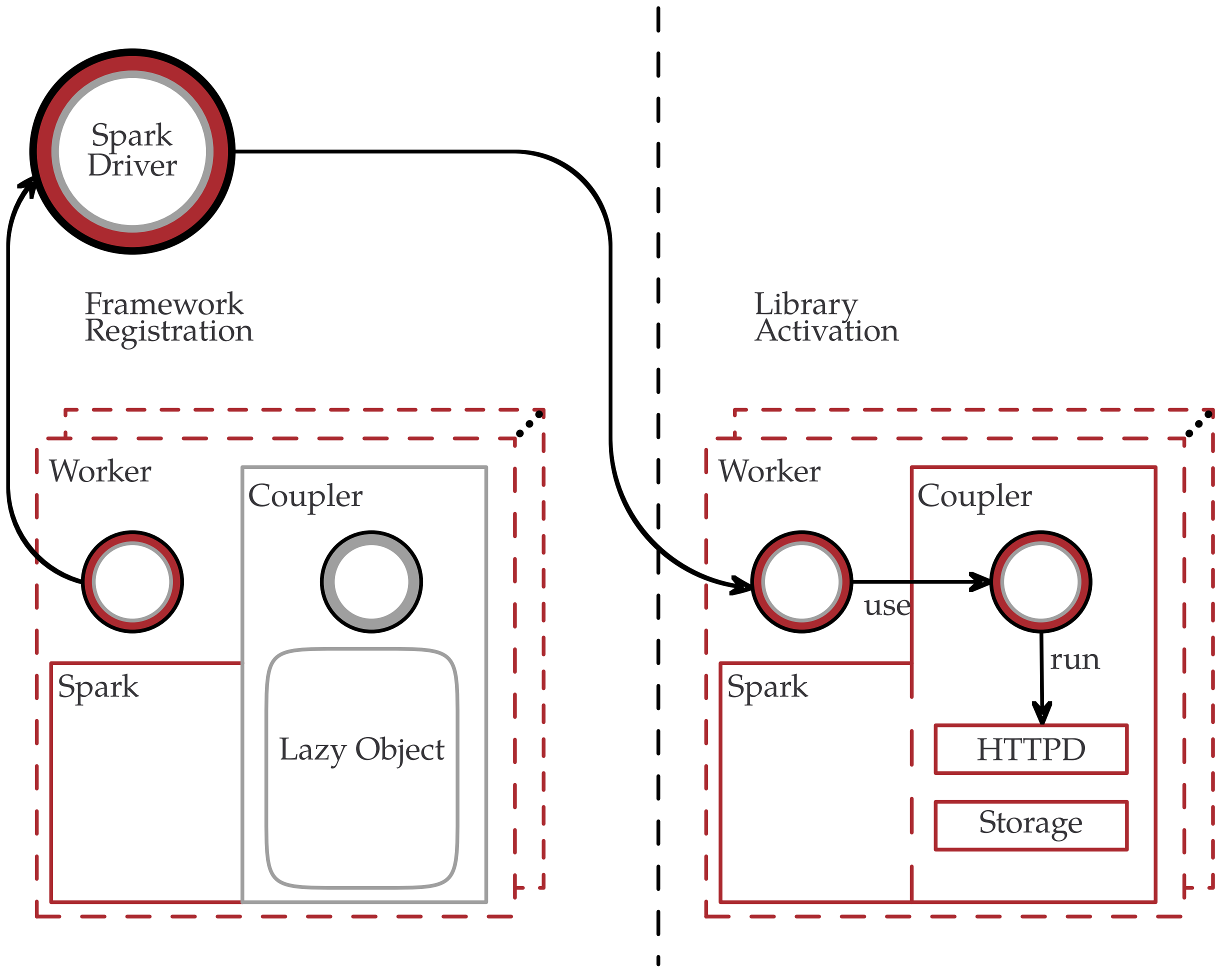

Figure 3.5 Because of Spark's implicit parallelism, the code for the coupler depicted here is part of every Spark process. As shown on the left, the actual object does not exist at first. When instantiated, the coupler contains the implementation of the transport described in the preceding sections.

Executor services (collectively labeled the *Coupler*) start running only after the driver accesses a lazy reference to the service object via a distributed function call (shown in **Figure 3.5**). This is, to our knowledge, the only way to provide services on Spark executors without modifying Spark itself. It relies on Scala's built-in semantics for lazy variables and the fact that Spark code which references lazy values instantiates them on the node where said code is actually executed.

Once executors' HTTP actors start, they enter a loop of waiting for metadata-bearing signals and handling them by assuming the consumer role from our transport protocol. Once metadata is received, the library needs to map POSIX shared memory and open some semaphores. It can map memory using a FileChannel, but POSIX semaphores have no built-in JVM support. Accordingly, our library includes a wrapper class which uses the foreign function interface (FFI) provided by the Java Native Runtime (JNR). JNR makes semaphore access simple: load the pthreads library and call the relevant sem_(open, wait, close, and unlink) methods. Once the library sets up its POSIX IPC constructs, it reads segments from the ring buffer and stores them in an array builder. This array builder will produce an iterator, upon request, as the source for a DataFrame. Each dataset has exactly one buffer which produces the concatenation of all rows transferred since the last time the buffer was cleared.

For distribution, we package our library code as a fat JAR file to minimize build complexity for end-user programs. We planned to distribute a source-only library because Spark uses multiple class loaders in a configuration which thwarts most attempts at dynamic class loading. Copies of the same class loaded by different classloaders are different to the JVM and cannot be coerced together. Fortunately, the pieces enabling this behavior are dropped by serialization; serializing and de-serializing a class from one loader can bring it into the proper context for use with another loader. This enabled us to move all of our application-agnostic code into a JAR library, simplifying both file management and the build process for end users.

3.4 RPC

In addition to implementing the producer and consumer halves of our transport protocol, our libraries also implement a simple RPC framework. The RPC framework is based on HTTP because it is simple and because we already had HTTP libraries involved for the metadata piece of the transport protocol.

The Spark driver, in addition to its other tasks, hosts an RPC service. This RPC service allows users to request analyses by name, and to provide a list of datasets upon which an RPC depends. This list of dependencies is used to register the data stored on the executors as a named table for use in Spark SQL; this allows users to access their data with simple SQL statements rather than more complex DataFrame operations that would invoke Scala's type system. Users are free, of course, to write more complex code. Initial data access is simpler for users, though, if they merely use ***SELECT * FROM <TableName>***. Doing otherwise requires either understanding our generated code, or writing replacement code for the generated file; both options are likely unpalatable for most users.

The output of the RPC service is more flexible than the input. Users are free to return anything they wish from their analysis methods, as long as it is serialized to byte form. The RPC system returns bytes specifically to provide as much freedom as possible.

Knowledgable users might wish to replace our simple RPC system with a more robust and feature-full system such as Thrift or gRPC. Such replacement would be relatively simple for any users with the interest; we designed it to be swappable because we anticipate the need of some users to have customized RPC which better suits their needs. The rest of our system, by contrast, is designed such that end-users would likely not care to modify it even though they could. Implementing a completely new data model would be the most likely motivation for anyone wanting to try, and the necessary changes would result in a near-complete overhaul.

4 Analysis

During initial testing, we used the standard Network Filesystem (NFS), but we eventually encountered the concurrency failure cases which make NFS unsuitable for this type of workload. From the Spark point of view, a filesystem which resembled HDFS would have been preferable, but such systems require local disks on compute nodes. This precludes their use in many HPC clusters, where compute nodes are often diskless, as well in our cluster, where we chose the same paradigm. Ultimately, we chose BeeGFS for storage because it approximates the state-of-the-art in HPC filesystems. Its architecture resembles that of Lustre, which would have been the ideal test system if not for its low stability and its correspondingly high maintenance requirements.

CPU	RAM	Ethernet	Infiniband	Storage	OS	Scheduler
Intel® Xeon® X5560	24 GB	Gigabit	QDR	BeeGFS	CentOS 7	Slurm

Table 4.1 Cluster Configuration

After sorting out our filesystem issues, our cluster's blades were configured as shown in **Table 4.1**. Our test cluster was a single rack of these blades, with one group set aside as BeeGFS storage and the rest registered with Slurm for use in compute jobs. Although all nodes had disk bays, only the storage nodes actually configured and used them. Most traffic ran over Gigabit Ethernet, but QDR Infiniband was configured and used to access BeeGFS.

4.1 Synthetic Benchmark

To establish basic performance data for our system, we wrote a synthetic workload which tested the features we most expect users would need. In particular, because our data model does not allow for nested structure, we chose to transfer two different but related datasets and perform a join within Spark. At first, we weren't sure what kind of analysis to run and settled for a simple count. In implementing the count, we encountered some errors, and the easiest way to debug and add a regression test was to write a Spark analysis which verified correct receipt of all data in both tables. We controlled test data amounts with Random123's Philox counter-based random number generators[15] to obtain repeatable tests which exercised Spark's analysis interface in a realistic fashion with reasonable coverage. The resulting test became our new synthetic benchmark as well as our main integration test.

Our fake data was designed to mimic some kind of generic molecular dynamics or computational chemistry application. Each node generated a rather large random number of atoms every timestep, and transferred two tables to Spark. The first table had timestep metadata, including the number of atoms intended to be in the second table. The second table had atoms, each with a label and position and velocity information, as well as a field indicating which timestep it belonged to.

```
SELECT  SUM(diff) FROM (
   SELECT  ex.rank, ex.total  -  got.total  AS diff
   FROM (
      SELECT rank, SUM(natoms) AS total
      FROM Timestep
      GROUP BY rank
   ) ex
   LEFT JOIN (
      SELECT rank, count(*) AS total
      FROM Atom
      GROUP BY rank
   ) got
   ON  ex.rank  =  got.rank
)
```

Figure 4.1 Benchmark SparkSQL

Both tables had a field indicating the rank of the MPI process which generated a particular row. Normally, applications wouldn't need or want such a field, but because our application was verification of correct transfers at node granularity, we needed to track data origins. The use of this field is clear from the actual SQL statement used in our test, shown in **Figure 4.1**; it's needed in two ***GROUP BY*** clauses.

The transfer to Spark was repeated ten times, after which the data size was increased and the process repeated. All data was re-generated with new random values for each iteration of the outer loop.

We were interested in transfer and analysis data times, so we measured them separately. For transfer time, we measured from just before the master started writing to just after an Mpi_Barrier placed after the write call. The write call was a function pointer whose actual implementation was different depending on which transfer mechanism we were measuring. Because our transfer mechanism was stateless, all setup and teardown was contained in the timing interval by default. For the filesystem, all resource opening and closing was likewise done inside the timed section.

4.2 Overhead

When first running our benchmark, we discovered unexplained crashes were reliably reproducible when our test data reached a certain size threshold. The discovered cause was simple: we were exceeding our compute nodes' memory limits. Normal Spark has the ability to swap when it runs out of memory, and we initially failed to account for the effect our embedded framework would have on Spark's deployment constraints.

To demonstrate that our framework was not causing an undue resource drain, we instrumented our framework and made copious JVM measurements. The results, shown in **Figure 4.2**, demonstrate that the ratio of overhead to the size of the data decreases to negligible levels as our simulated datasets become realistically large.

Our tests show that our framework is not leaking memory or causing other unwarranted behavior. They also provide a basis for estimating how much RAM to allocate for a coupled workflow, or how large a workflow can be run given a known set of resources. Most extra memory used by our system over pure Spark is an artifact of the need to copy between two independent programs. The inability to swap data out is due to our custom data structure not being a file and thus not having the same persistence abilities as other Spark data sources.

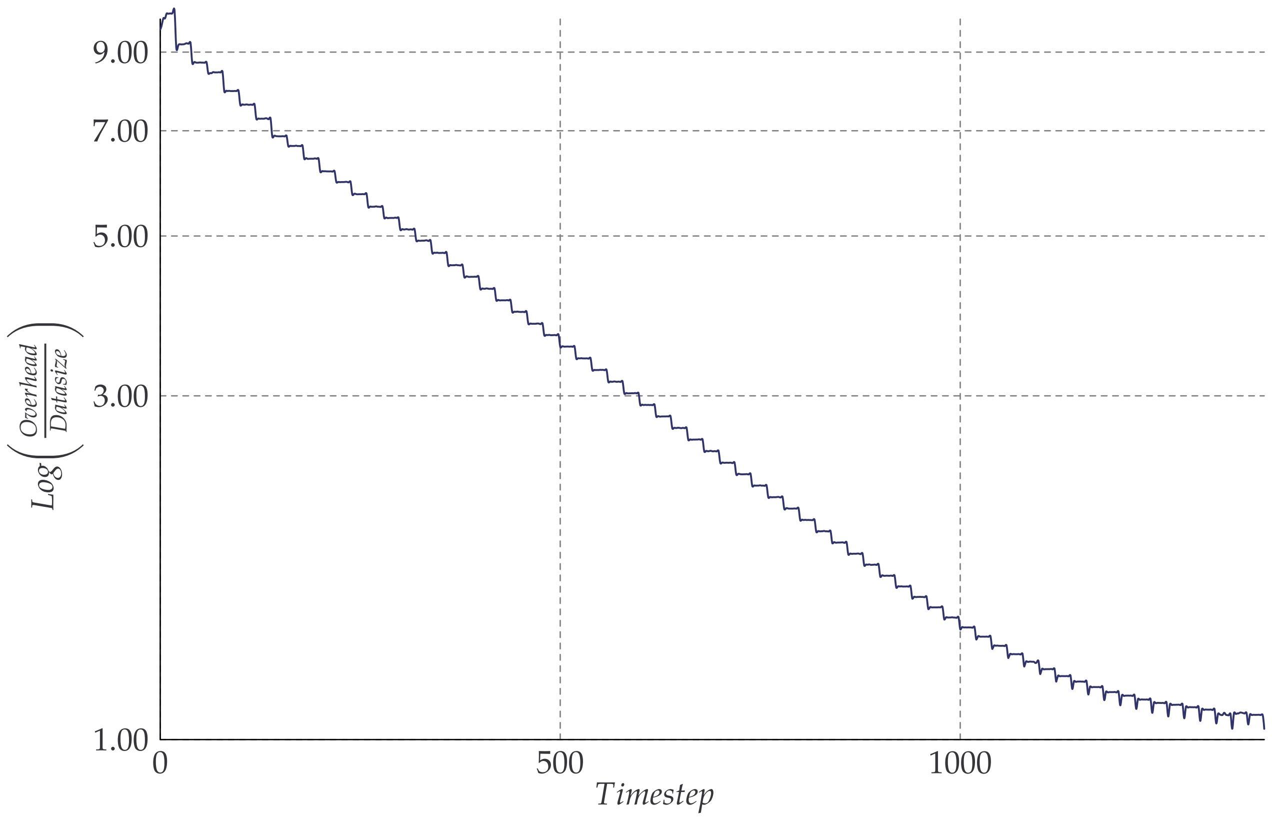

Figure 4.2 Ratio of Overhead to Data Size for Increasingly Large Data Steps

Our coupled codes need at least three times as much memory as just the simulation code alone. This may seem high, but Spark alone likely uses more memory than the simulation code. The baseline we are comparing with, then, is already somewhere between two and three times the memory of a standalone simulation. As a result, our library itself must uses less than one simulation's worth of memory. By design, it should use a predictable amount which is far less. Spark's memory usage, however, increases during certain execution stages and can be hard to predict. Because Spark has some lazy behavior, larger-than-possible jobs may appear to work for a time and mysteriously crash when the fatal allocation is finally attempted. Since this can cause difficulties in debugging, we recommend that users carefully pre-calculate their expected resource usage, and that they err on the side of over-allocation.

4.3 Results

We ran our benchmark with a range of sizes from ridiculously small to as large as would fit on a node without crashing from out-of-memory errors. For every size step we ran the test ten times. Also, due to the local nature of our transfer library, we were able to collect results from many pairs of Spark and native processes running concurrently on separate nodes. This gave us a large number of samples for each step. We computed summary statistics over these, and discovered several results.

- The first run in a program coupled with Spark should probably be thrown out because the system's initialization causes an enormous outlier. This was the case regardless of which transfer system we used.

- There is little benefit to our system if data sizes are trivial. This is apparent from **Figure 4.3**, where small values of data have total overlap in times. If anything, our system's setup overhead takes it just a small amount longer than vanilla Spark, making it detrimental if you only compare initial startup speeds. These cases where the filesystem method achieves lower speeds are more obvious when examining **Figure 4.4**.

- When real data starts moving, our system quickly takes and maintains a significant lead over Spark-on-BeeGFS. The practical difference at the scale we were able to test is

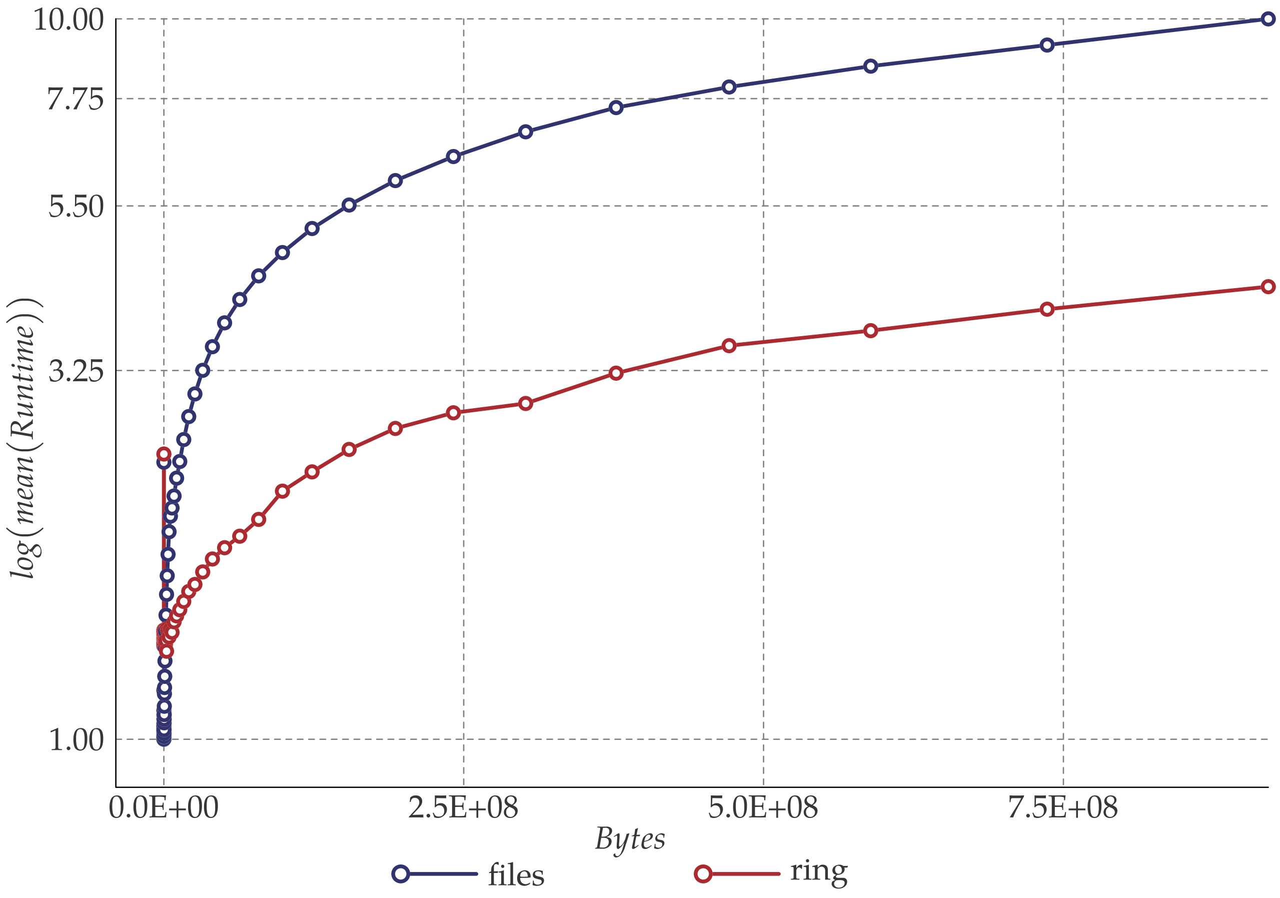

Figure 4.3 Comparing Filesystem (files) and Ring-Buffer (ring) Transports: Runtime with Increasing Size

between jobs measured in minutes or seconds and jobs measured in hours. If systems have enough memory, we expect the scaling would extend to enable job completion in minutes or hours instead of days or weeks. As it is based on risky extrapolation, the previous suggestion is unreliable for many purposes, but it does provide a likely upper bound on the utility of our system. The possibilities permitted by this bound should be enough to aid in getting future tests scheduled on large systems with enough resources to properly explore the practical limits of our approach.

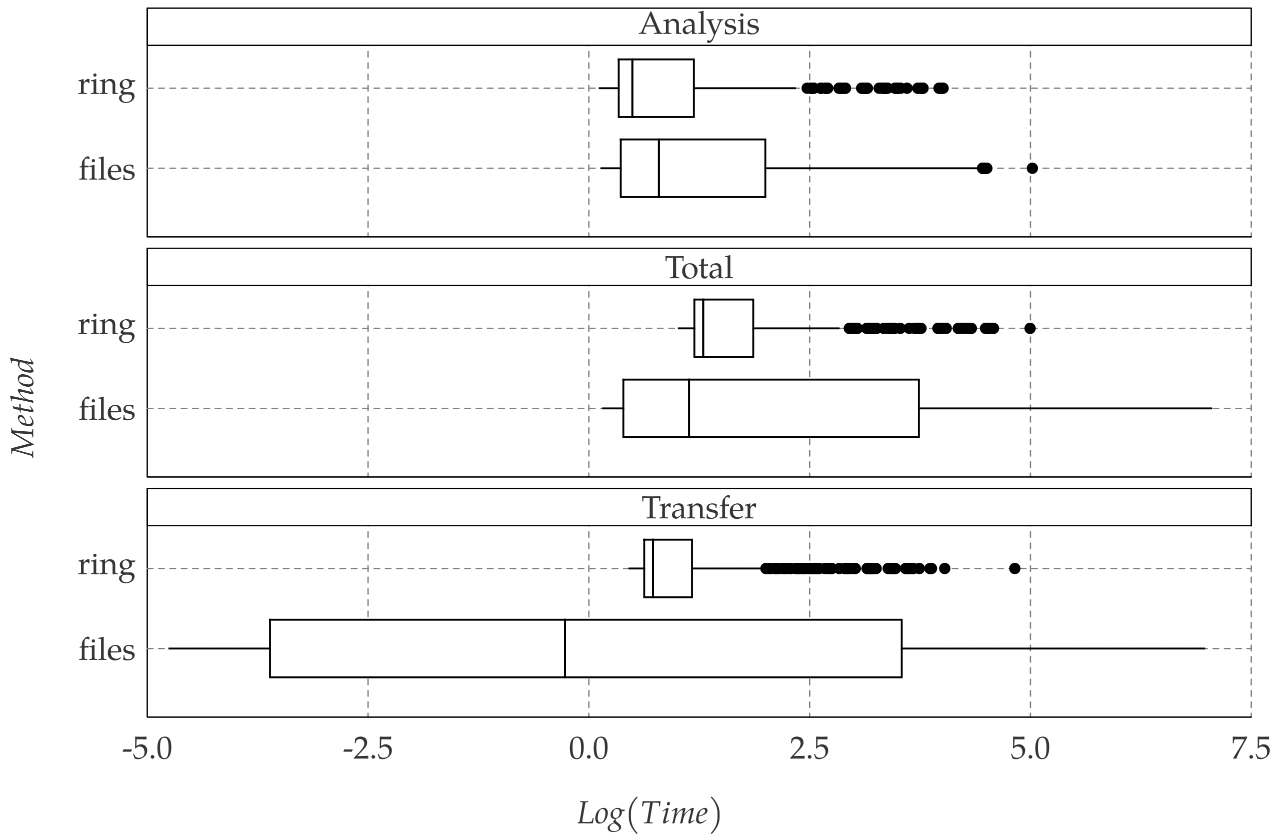

Figure 4.4 Comparing Filesystem (files) and Ring-Buffer (ring) Transports: Time Distribution

Other than the overall scaling of our approach, we wanted to know how it affected the major phases of a coupled workflow. As seen in the boxplots of **Figure 4.4**, we separately recorded the analysis and data transfer phases, and examined the distributions from both methods. Note the presence of numerous outliers in the ring transport's timing distribution; multiple factors could contribute to this.

- The ring buffer times increase slowly and thus our sample range contains many similar results. The far end of the range is just starting to use data sizes which result in significantly larger times. As size grows, though, our sampling density decreases and we have few enough data points to result in the last samples appearing as outliers against the distribution of the more densely sampled low-runtime region.

- Our in-memory transport seems to be quite predictable, with a low variance in performance. However, our compute nodes are diskless and all system services use a ram-based filesystem. This creates a chance of the transport on a single node being drastically affected, which would cause an outlier in our data. As we increase data size and saturate nodes' memory, this problem could become more common. The cluster filesystem is unaffected by such problems, which would help explain the relative lack of outliers in filesystem results.

Given that large data is the source of both factors we identified, the outliers are likely an artifact of running our tests up to the memory limits of our test cluster.

Our contribution is targeted at accelerating transfers between simulations and Spark, so we would expect the transfer phase to be accelerated. This is exactly what we observed. Further, we did not observe a significant change in analysis time between the two methods. In summary, we maintain normal Spark analytics performance while vastly improving data ingestion.

4.4 Proof of Concept

Our benchmarks show, quantitatively, that our work enables novel interactions between Spark and native HPC codes. However, while our benchmark was intended to resemble a simulation code, it was also a benchmark and a system test, so it may leave some uncertainty about how to apply the new interactions we've enabled. To address this uncertainty, we developed a qualitative test of our library interfaces, showing how a native system using a typical mix of C[16] and Fortran[17] can interact with Spark and its ecosystem.

Our example system performs adaptive Monte-Carlo sampling, and was inspired by VEGAS [18]. The basic task is that of finding the shape of an unknown function on a bounded Cartesian plane. The function we evaluate is created by fitting a fourth-order spline to a random walk along the y-axis and an equally spaced set of integer x-coordinates.

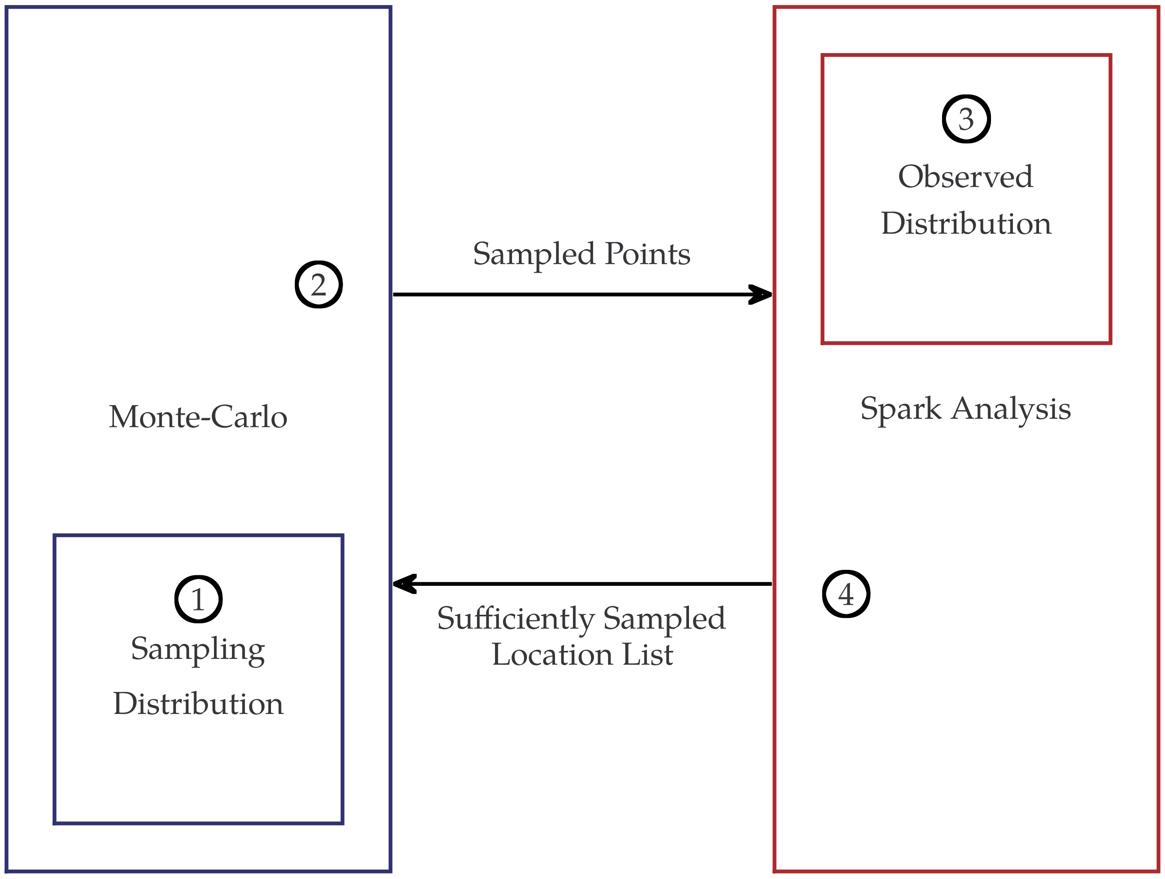

Figure 4.5 Demo Overview

In **Figure 4.5**, we display the high-level setup of our demo system. The figure's labeled areas are explained in more detail below.

1. The **sampling distribution** is calculated from a two-dimensional histogram which tiles the search plane into equal size blocks, as specified in **Figure 4.6**. Initially, this histogram is uniform, but can be updated using data produced in item 4 of **Figure 4.5**.

Histogram

$$\begin{array}{lll} \mathit{bins} : & \mathrm{seq} & \mathbb{R} \\ \mathit{binsPerDim} : & \mathbb{Z} & \\ \mathit{count} : & \mathbb{Z} \to & \mathbb{R} \end{array}$$

Search Plane

$$\begin{array}{ll} x_0, y_0, \mathit{width}, \mathit{height} : & \mathbb{Z} \\ \mathit{valid} : & \mathbb{R}^2 \to \{\mathit{true}, \mathit{false}\} \end{array}$$

$$\begin{array}{ll} \forall\ (x,y) : \mathbb{R}^2 \bullet \mathit{valid}\ (x,y) & = \\ \quad x_0 \leq x < x_0 + \mathit{width} & \wedge \\ \quad y_0 \leq y < y_0 + \mathit{height} & \end{array}$$

Init

Histogram
Search Plane

$$\begin{array}{l} x_0 = y_0 = 0 \\ \mathit{width} = \mathit{height} = 100 \\ \mathit{binsPerDim} = 64 \\ \mathit{bins} = \left\{(i, 100) \mid i \in 0..\mathit{binsPerDim}^2\right\} \end{array}$$

Tiling

Histogram
Search Plane

$$\begin{array}{lll} \mathit{binwidth} : & & \mathbb{R} \\ \mathit{bin} : & \mathbb{R}^2 \to & \mathbb{Z} \end{array}$$

$$\mathit{binwidth} = \frac{\mathit{width}}{\mathit{binsPerDim}}$$

$$\begin{array}{l} \forall\ (x,y) : \mathbb{R}^2 \bullet \mathit{valid}\ (x,y) \implies \\ \quad \mathit{bin}\ (x,y) = \mathbf{let}\ (i,j) = \left\lceil \dfrac{(x,y)}{\mathit{binwidth}} \right\rceil \bullet \\ \quad (i \cdot \mathit{width}) + j \end{array}$$

Sampling Distribution

Tiling

$$\begin{array}{lll} \mathit{probability} : & \mathbb{R}^2 \to & \mathbb{R} \\ \mathit{draw} : & \mathbb{N} \to & \mathrm{seq}\ \mathbb{R}^2 \end{array}$$

$$\begin{array}{l} \mathbf{let}\ \mathit{area} = \mathit{binwidth}^2 \bullet \\ \forall\ (x,y) : \mathbb{R}^2 \bullet \mathit{valid}\ (x,y) \implies \\ \quad \mathit{probability}\ (x,y) = \dfrac{\mathit{count}\ (\mathit{bin}\ (x,y))}{\mathit{sum}\ (\mathit{bins}) \cdot \mathit{area}} \end{array}$$

Figure 4.6 Both halves of the demo use the same model for histograms discretizing a search space. The monte-carlo search process initializes its histogram to a uniform positive value and uses it to drive its sampling distribution. The given parameter values correspond to the output graphs later in this section.

An adaptive program which uses this updating strategy can also choose to use the sum of this histogram as criterion for early termination; it can stop, for example, if all the bins have been updated. This stopping condition, labeled Swept, is described (along with the update procedure) in the lower-right corner of **Figure 4.7**.

2. At each simulation step, the native library samples numerous points from the distribution in item 1. It evaluates these points and labels them with 1 if they fall strictly under the target curve, and -1 otherwise.

Spark Client

$analyze : \quad \text{seq SAMPLE} \rightarrow \text{seq } \mathbb{Z}$

Spline

$\overline{x}, \overline{y} : \quad \text{seq } \mathbb{R}$

$eval : \quad \mathbb{R}^2 \rightarrow \text{SAMPLE}$

Parameter

$step_{max} : \quad \mathbb{Z}$

$npoints : \quad \mathbb{N}$

Program

Parameter

Sampling Distribution

Spark Client

Spline

$step : \mathbb{Z}$

Init

Program

$step = 0$

$step_{max} = 1000$

$npoints = 1000$

$\overline{x} = \{i \mid i \in \mathbb{Z} \wedge i \in 0..100\}$

$\overline{y} \subset WALK$

$\# \overline{y} = \# \overline{x}$

$\text{SAMPLE} == \mathbb{R} \times \mathbb{R} \times \{1, -1\}$

$\text{WALK} == \text{seq } \mathbb{R}$

Exhausted

ΞProgram

$step = step_{max}$

Iterate

ΔProgram

$step : \mathbb{Z}$

$feedback : \text{seq } \mathbb{Z}$

$step? < step_{max}$

$step' = step + 1$

$feedback = analyze(eval(draw(npoints)))$

Swept

ΞProgram

$sum\,(bins) = \#bins$

Adaptive Step

Iterate

ΔHistogram

$sum\,(bins) > \#bins$

$bins' = bins \oplus \{(i, 1) \mid i \in feedback\}$

Figure 4.7 Spark feedback can be used to update the probability distribution. While a more complex model is easily enabled by letting Spark choose new values for each bin, this simple version merely sets bin values to one when Spark returns their indices.

The x and y coordinates as well as this label are sent to Spark for all the samples from the current round, according to the interface in **Figure 4.7**.

3. The **observed distribution** which Spark maintains is a histogram with the same dimensions as the one in item 1, but is initialized with empty bins.

Observed Distribution
Tiling

Init
Observed Distribution

$binsPerDim = 64$
$bins = \{(i, 0) \mid i \in 0..binsPerDim^2\}$

Update
ΔObserved Distribution
$sampledPoints? : \text{seq } SAMPLE$

$$bins' = bins \oplus \{ \\ (bin(x, y), count(bin(x, y)) + v) \\ \mid (x, y, v) \in sampledPoints? \\ \}$$

Figure 4.8 Updating Observed Distribution

Spark updates its histogram with the data described in item 2 according to the model in **Figure 4.8**. At the end of the program, this histogram is the one written to disk and plotted for analysis.

4. In addition to updating its persistent histogram, Spark also calculates a per-round histogram and weights the bins using data from its persistent histogram. This temporary histogram is identical to the one described in **Figure 4.8**, but the initialization step is performed every round instead of once at the beginning of a long program.

Historical Bin Weighting

$$\begin{array}{ll} past?, current? : & \mathrm{seq}\ \mathbb{R} \\ weighted! : & \mathrm{seq}\ \mathbb{R} \end{array}$$

$$\begin{array}{l} \mathbf{dom}\ past? = \mathbf{dom}\ current? \\ weighted! = \{\ (i, ||p| + c|) \\ \quad \mid (i, p) \in\ past? \land (i, c) \in\ current? \\ \} \end{array}$$

Sufficiently Sampled

$$\begin{array}{ll} weighted? : & \mathrm{seq}\ \mathbb{R} \\ threshold? : & \mathbb{N} \\ feedback! : & \mathrm{seq}\ \mathbb{Z} \end{array}$$

$$\begin{array}{l} feedback! = \{i \\ \quad \mid (i, v) \in\ weighted? \land v > threshold? \\ \} \end{array}$$

Figure 4.9 Histogram Feedback

The weighting and thresholding rules we used are summarized in **Figure 4.9**, but they can be reconfigured. Cells with values exceeding some threshold are considered **sufficiently sampled**, and their locations are sent in response to the RPC request. If it so chooses, the native client can update the histogram for the **sampling distribution** (item 1) to effectively stop sampling these locations and focus on other areas of the plane.

4.4.1 Spark Analysis

We chose one representative library to demonstrate the power of Spark's ecosystem: SystemML[19]. SystemML augments Spark's numeric matrix processing and enables a mix of dynamic scripting with compiled Spark code. While it also has numerous libraries of its own, its dynamic script loading is the most interesting feature when combined with our library's simple HTTP RPC interface. This combination allows simple embedding of scriptable network APIs into workflows to simplify the bulk of the application code.

The Spark half of our program relies on mutable driver state to track what happens over time. SystemML-computed values are extracted from its symbol table and stored. These values are iteratively updated by registering the same symbol for input and output, and passing in the stored value as input. The symbol table is initially empty, so SystemML does require an initialization phase. An easy way of dealing with this is to simply make a separate handler function that (re-)initializes the symbol table. A more clumsy solution could involve editing the script run by the analysis handler to include the initialization code, running one iteration of the workflow, and reverting the script for future iterations.

SystemML and Spark are adept at different things. Calculation of histogram updates and many other numeric array operations are easily expressible with SystemML, but if a list needs to be sorted, grouped, or manipulated by myriad other functional operators, Spark SQL and Scala are the only sensible recourse. In the end it seems best to have Spark deal with parsing, reshaping, filtering, sorting, encoding, and other data manipulations for input and output to simulation processes. SystemML, meanwhile, should be left to do the most dynamic operations. Anything which is likely to change and can be expressed in the form of a SystemML script probably should be. Output for debug or statistical purposes is a special case of dynamic operation which SystemML excels at, and having a zero-dependency zero-return method that takes all state variables as inputs and calls an output script is a simple way of performing on-demand introspection into Spark's system without affecting unrelated components.

4.4.2 Glue

Adding scripting to a system is often a major undertaking, but scriptable systems are typically easier to prototype with. To clarify, here scripting denotes a system of dynamic code loading and execution with hooks to a compiled program's interfaces. SystemML provides an interface through which code resembling R or Python can be loaded at runtime from arbitrary files. While the system doesn't support most of the standard libraries of its syntactic lookalikes, it does have a number of its own libraries, and it uses a distributed matrix model that is stable and performant because it is based on Spark.

We did not include a scripting interface with our library, but here we demonstrate how to add one without modifying any native code. Because we used HTTP for the RPC analytics interface, any language with an HTTP client can interact with it. When writing the Scala code for a workflow, arbitrary methods can be inserted purely for interacting with a script. As the Spark driver is capable of storing and manipulating state, it can be convenient to have setup methods called by a script before launching a simulation. Likewise, writing Spark's state to a file can be part of a post-run script. This makes it easier to change the internal behavior of the analytics without affecting how interaction happens with a simulation. Many simulations have heavy limitations on modification of their binaries, so this embedding of a flexible interface has significant potential.

4.4.3 Output

We ran two versions of our program. One, which we call *adaptive*, modified the histogram based on information polled from Spark at the end of each timestep. The other, dubbed *exhaustive*, never modified the histogram. We called the second exhaustive because the first method was allowed to stop once Spark's replies had updated all the histogram cells. In our demonstration run, the adaptive method was able to finish in 23 iterations instead of running for a full 1000 like the exhaustive version. We did not collect statistics on the exact advantage the method has; that is not the focus of this section. Rather, we direct attention to what information this much shorter runtime was able to collect, compared to its exhaustive counterpart.

In **Figure 4.10**, we show the two-dimensional histograms which our Spark process calculated. Note that the curve discovered is nearly identical, with the main difference being the scale of values in the histogram bins.

More complex decision criteria can be implemented with few-to-zero changes to the interfaces between system components. If interface changes are needed in a program such as this, our system can support them because the main infrastructure doesn't care what high-level behavior is running atop it. Every bit of code in this example is *user code*, and thus end users have full control over it.

Compare this to traditional analytic workflow engines; many libraries have a version of the method we implemented. In fact, we could have avoided using Spark at all and simply used built-in functions from the GNU Scientific Library. Once we wanted to do

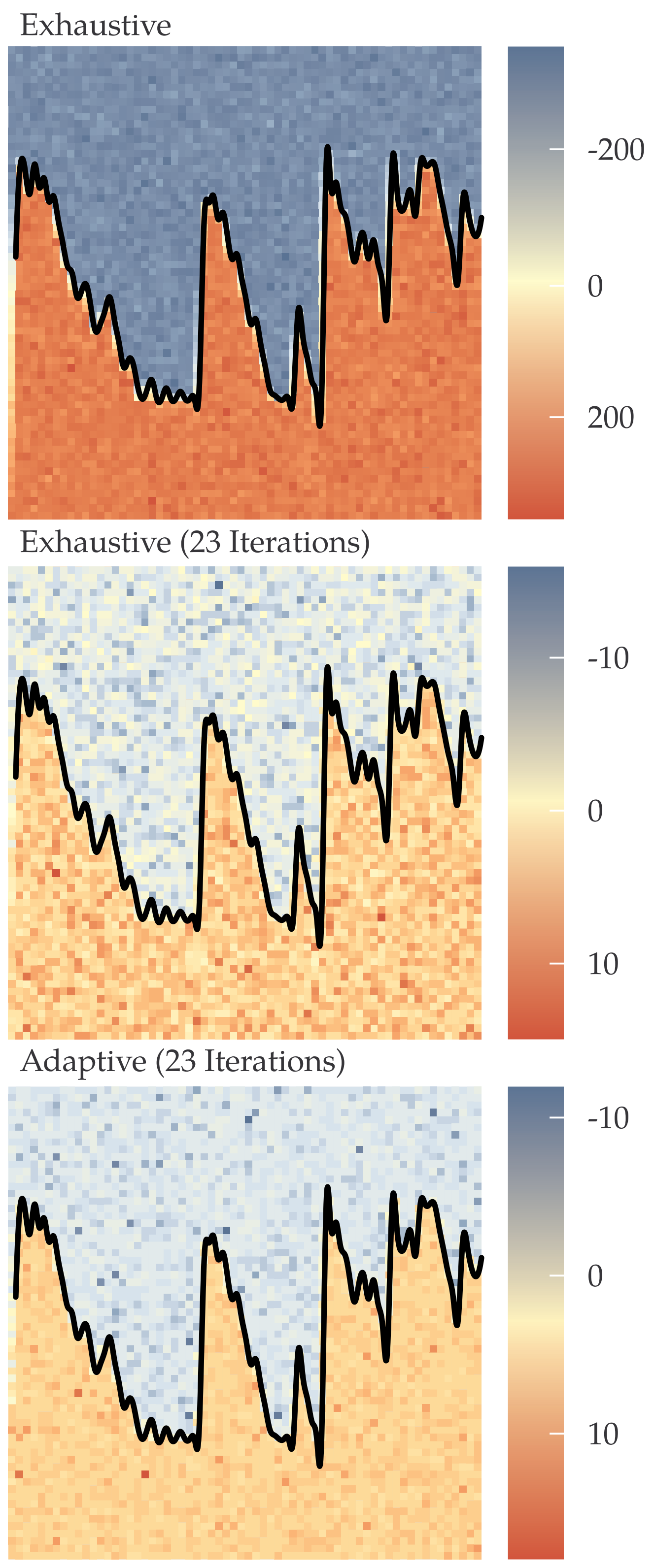

Figure 4.10 Adaptive Monte Carlo quickly terminates while maintaining good space coverage. Non-adaptive programs of the same length produce noisy, low-quality output.

something new, and modify our system, we would be stuck if the monolithic libraries did not provide built-ins to support our new ideas. So although our simple code is not a complete replacement for any given library, it provides many of the benefits with none of the traditional costs of library usage.

5 Conclusion

Apache Spark and its ecosystem offer great potential to enhance the development of simulation analysis workflows. Spark seems nearly incompatible with other parallel frameworks, and most Spark interactions must use the filesystem because guarantees on other interfaces are scarce. Fortunately, Spark's implementation language, Scala, allows a workaround.

We leveraged Scala's lazy semantics to embed, into executors, services which listened for metadata using local sockets and which transferred datasets from shared memory into local storage buffers. This allowed simulation processes to efficiently inject data into executors. By concatenating the resulting rows and registering them as virtual SQL tables, we enabled Spark to analyze live simulation output as normal data frames. We gave coupled simulations access to analyses conducted on these data frames by registering analysis methods with a simple RPC-over-HTTP interface.

With these few manipulations, we obtained a library which completely bypassed the filesystem. By comparison with a state-of-the-art cluster filesystem, we showed that our method's performance significantly exceeded the previous threshold for this type of coupled program. We also showed, through profiling, that while our method introduces additional memory overhead, the overhead is not unreasonable.

We showed that not only is our library performant; it also meets its usability goals by making analysis codes easy to write and maintain. Using Monte-Carlo search as a representative example, we showed that adaptive codes can easily be expressed in our framework. Similar programs already exist, but are implemented in user-unmodifiable libraries and cannot be easily customized for different client applications. In contrast, our framework places the whole algorithm in user code; it thereby facilitates simple and even dynamic updates to key parts of the algorithm. By combining dynamic updates with our RPC-over-HTTP system, live script-based analysis can be integrated with arbitrary workflows for ultimate flexibility.

Our build and distribution systems are also tuned for usability. We packaged our components in convenient library formats for users to include and link with. Users need only write normal simulation code with a few added library calls, and normal analysis codes conforming to a simple interface. We even simplify launching coupled codes with a helpful supervisor framework for use with Slurm clusters.

In summary, our contributions make Spark's potential a reality and demonstrate some of the power that Spark brings to scientific workflows. Of course, our system is not perfect. We took note of outstanding problems which we encountered and had to work around. We discuss the most prominent drawbacks along with our recommendations for future research using our system.

5.1 Future Work

Projects to improve on the work in this paper could extend the data model, making the APIs more flexible on the simulation side. They could also enhance the RPC mechanism for getting results back. For example, they could craft a system where all nodes can independently fetch results without involving the others. That would likely require broadcasting data back down into the executors and setting up an automatic trigger that bypasses the need for explicit requests to execute analysis tasks. Regardless of their purpose, future projects in this area should consider the versioning problem we encountered.

5.1.1 Versioning Issues

It cannot be overstated how badly version-locked the Apache data ecosystem has become. The symptom is partly due to general trends in JVM development, but is exacerbated by projects' imprecise API management, which leads to frequent breakages even with minor version shifts. As a result, most projects only officially support the version of tools that existed when their code was originally developed.

We took great care to avoid this problem in our own project, but found it impossible to escape entirely. While our project's core functions are flexible with regard to Scala and Spark versions, libraries like SystemML are not. This effectively defeats the goal of leveraging an existing machine learning ecosystem, because most of that ecosystem is frozen in the past.

Besides the inability to simultaneously keep up with Spark development and use its ecosystem, we encountered issues with the maintenance of foundational tools. The Java Native Runtime (JNR) and a common Java serialization library both use code generation, but their versions are not necessarily compatible. There are records of breakages caused by this incompatibility, but the error messages are vague enough that other bugs could be confused with it. We discovered this behavior when JNR refused to work with the latest Spark and Scala versions. Because we couldn't isolate the cause, we had to rewrite the semaphore code using the Java Native Interface (JNI) and C directly.

Were we to adopt that change permanently, it would mostly kill the utility of a JAR library and add more complexity to Spark environment setup. We only experimented with JNI to verify that our library was otherwise portable to Spark 2.4.3 and Scala 12.8; we also modified SystemML for the same purpose. While working on updating SystemML, we noted the library and build system are badly designed for use with Spark and Scala, instead relying on half-converted Java code. A proper rewrite would simplify many things, without requiring overly much time from proficient Scala developers. Thus, the greatest barrier to SystemML being up-to-date is ultimately a lack of time or interest from properly skilled developers.

We managed a forced upgrade of dependencies within a few days, but we suggest that a change of mindset in this community may be necessary for interoperability to have a future.

5.1.2 Future

Even assuming the versioning issues become more manageable, there are limits to how far this work can go. However much improvement is made to this system, the performance data we collected demonstrates that there will always be drawbacks to coupling two disparate systems this way. Node memory is effectively halved at best due to maintaining a copy for each system. Also, communication times still have a high penalty; copying memory into transport and back is obviously slower than not copying memory at all. The interprocess barrier is ultimately the biggest hindrance to any coupled-process workflows.

Ultimately the best use of our system is probably in designing its own replacement: a system that is process-native and truly integrated with simulation codes, not forcibly divorced from them. Our system can serve as a prototype to help select functionality that any new system should have. It also provides a minimal API that a replacement could implement, which facilitates testing. Our project could also provide some code-generation utilities that convert prototype methods into useful native libraries. With the ongoing development of Scala's LLVM backend, it is possible that the main development language could even remain the same.

Obviously, much work is needed to build native libraries with feature parity to the ones in the Apache ecosystem, but given that this particular line of research ignores the various fault-tolerance schemes used in such systems, building comparable systems is potentially far simpler than building the originals. By leveraging existing HPC suites in new ways, a suitable system might ultimately be assembled without writing much more than glue code.

Bibliography

[1] Argonne National Laboratory, *Data-Intensive Science*, http://www.mcs.anl.gov/group/data-intensive-science (unpublished). (Accessed: 2017-03-21)

[2] A.J. Pawel and G.L. Mesina, *Uncertainty Analysis for RELAP5-3D*, Technical report (Idaho National Laboratory (INL), 2011).

[3] B. Silva, M.A.S. Netto, and R.L.F. Cunha, JobPruner: A machine learning assistant for exploring parameter spaces in HPC applications, *Future Generation Computer Systems* **83** 144–157 (2018).

[4] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin *et al.*, Apache Spark: a unified engine for big data processing, (2016).

[5] B.M. Adams, W. Bohnhoff, K. Dalbey, J. Eddy, R. Hooper, K. Hu, L. Baum, and A. Rushdi, DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: version 6.3 user's manual, *Sandia National Laboratories, Tech. Rep. SAND2014-4633* (2015).

[6] Apache Software Foundation, *Apache Ignite*, https://ignite.apache.org/ (unpublished).

[7] H2O.ai, *H2oai/sparkling-water*, https://github.com/h2oai/sparkling-water (unpublished). (Accessed: 2017-03-21)

[8] OpenCPU, *Onetapbeyond/opencpu-spark-executor*, https://github.com/onetapbeyond/opencpu-spark-executor (unpublished). (Accessed: 2017-03-21)

[9] B. Mozafari, J.R.S. Menon, Y.M.S.C.H. Bhanawat, and K. Bachhav, SnappyData: A unified cluster for streaming, transactions, and interactive analytics, In 8th Biennial Conference on Innovative Data Systems Research (CIDR 17) (2017).

[10] T.G. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, High-performance language interoperability for scientific computing through Babel, *The International Journal of High Performance Computing Applications* **26**(3), 260-274 (2012).

[11] S. Palkar, J.J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab, Weld: A Common Runtime for High Performance Data Analytics, In 8th Biennial Conference on Innovative Data Systems Research (CIDR 17) (2017).

[12] Dask Development Team, *Dask: Library for dynamic task scheduling*, http://dask.pydata.org (2016). (Accessed: 2017-03-21)

[13] J. Computing, Big time series analysis with JuliaDB, *Wilmott* **2018**(95), 10–13 (2018).

[14] G. Ostrouchov, W.C. Chen, D. Schmidt, and P. Patel, *Programming with Big Data in R*, http://pbdr.org/ (2012). (Accessed: 2017-03-21)

[15] J.K. Salmon, M.A. Moraes, R.O. Dror, and D.E. Shaw, Parallel random numbers, In SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (ACM Press, 2011).

[16] M. Galassi *et al.*, *GNU Scientific Library Reference Manual (3rd Ed.): for GSL version 1.12* (Bristol: Network Theory, 2009).

[17] J. Williams, *Bspline-Fortran: Multidimensional b-spline interpolation of data on a regular grid*, Zenodo, doi: 10.5281/zenodo.1215290 (2018).

[18] G.P. Lepage, A new algorithm for adaptive multidimensional integration, *Journal of Computational Physics* **27**(2), 192–203 (1978).

[19] A. Ghoting, R. Krishnamurthy, E.P.D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, SystemML: Declarative machine learning on MapReduce, *2011 IEEE 27th International Conference on Data Engineering* 231-242 (2011).

APPENDIX A Spark Demo Program

This is code from the qualitative analysis in Chapter 4. It includes operations which make similar programs easier to write, but which is uninteresting on its own (eg. boilerplate), and thus is not included in the main document. Still, it is novel code facilitated by our library and thus placed here for completeness. The corresponding native code is not novel or interesting, so we omit its full version entirely; this appendix only contains Scala code and SystemML scripts.

A.1 Spark Driver Code

```
import edu.byu.csl.courier.service.SimpleRPC
import com.typesafe.config._
import collection.JavaConverters._
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.sysml.api.mlcontext._
import org.apache.sysml.api.mlcontext.ScriptFactory._
import scala.collection.mutable.{Map => MutMap}

trait HandlerFactory[A] {
  def apply(spark: SparkSession): Array[Byte]
}

object HandlerFactory {
  implicit val f1 = new HandlerFactory[Initializer] {
    def apply(spark: SparkSession) = new Initializer(spark).apply
  }
  implicit val f2 = new HandlerFactory[UpdateHandler] {
    def apply(spark: SparkSession) = new UpdateHandler(spark).apply
  }
  implicit val f3 = new HandlerFactory[Printer] {
    def apply(spark: SparkSession) = new Printer(spark).apply
  }
  def apply[A:HandlerFactory]: HandlerFactory[A] =
    implicitly[HandlerFactory[A]]
  def applyMap(handlers: Map[String,HandlerFactory[_]]) =
    handlers.map { case (k,v) => (k, v.apply(_)) }
}

object State {
  var symT: MutMap[String,Matrix] = MutMap()
  def update(k: String, v: Matrix) = symT.put(k,v)
}

class ScriptPlus(name: String)(implicit val spark: SparkSession) {
  val mlctx = new MLContext(spark)
  val conf = ConfigFactory.load("pruner")
  val namespace = "systemml.dml.scripts"
  val path = conf.getString(s"$namespace.$name.path")
  val intentOut = conf.getStringList(s"$namespace.$name.out").asScala
  var script = ScriptFactory.dmlFromFile(path).out(intentOut)

  def chain(f: () => Script): ScriptPlus = { script = f(); this }
  def in(name: String, data: DataFrame) = chain(() => script.in(name,data))
  def in(st: MutMap[String,Matrix]) = chain(() => script.in(st))
  def asVal() = chain(() => script.out(name)) //Name = return value
  def withDf() = in("df", spark.sql("SELECT * FROM Point"))
  def withState() = in(State.symT)
  def execute() = mlctx.execute(script)
}

abstract class Handler(spark: SparkSession) {
  implicit val session = spark
  val state_vars = new ScriptPlus("init").intentOut

  //Abstract
  def apply(): Array[Byte]

  //Shared
  def voidHandler(f: () => Unit) = { f(); "()".getBytes() }
  def updateState(res: MLResults) = state_vars.foreach { v =>
    State.update(v,res.getMatrix(v))
  }
  def dmlScript(name: String) = new ScriptPlus(name)
}

class Initializer(spark: SparkSession) extends Handler(spark) {
  def apply() = voidHandler(()=>updateState(dmlScript("init").execute))
}

class UpdateHandler(spark: SparkSession) extends Handler(spark) {
  def apply() = {
    val res = dmlScript("analysis").withDf.withState.asVal.execute
    updateState(res)
    res.getMatrix("analysis").toDF().createOrReplaceTempView("Hist")
    val saturated = spark.sql("""
      SELECT * FROM (
        SELECT __INDEX as Idx, ABS(C3) as Sat
        FROM Hist
      )
      WHERE Sat > 3
      ORDER BY Sat DESC
      """
    )
    saturated.collect().map(_.mkString(" ")).mkString(" ").getBytes()
  }
}

class Printer(spark: SparkSession) extends Handler(spark) {
  def apply() = voidHandler(()=>dmlScript("printer").withState.execute)
}

object Main extends App {
  SimpleRPC.run("0.0.0.0", HandlerFactory.applyMap(Map(
    "Init" -> HandlerFactory[Initializer],
    "Update" -> HandlerFactory[UpdateHandler],
    "Printer" -> HandlerFactory[Printer]
  )))
}
```

A.2 SystemML Scripts

Note that for these scripts and the preceding Scala code to work together, an appropriate *application.conf* is needed. The configuration file contains the locations and out-parameter lists for of each script, and should look like the following:

```
systemml.dml {
  scripts {
    init {
      path = "<dml path>/init.dml"
```

```
          out = [
             h
          ]
       }
       analysis {
          path = "<dml path>/analysis.dml"
          out = [
             h
          ]
       }
       printer {
          path = "<dml path>/printer.dml"
          out = [
          ]
       }
    }
}
```

A.2.1 Initialization

```
nbins = 64
h = matrix(0,nbins,nbins)
```

A.2.2 Histogram Update

```
nbins = nrow(h) #for square space
dmin = 0
dmax = 100
stride = dmax/nbins
cuts = seq(dmin,dmax,stride)
x=df[,1]
y=df[,2]
b=df[,3]
gx=ceil(x / stride)
gy=ceil(y / stride)
analysis = matrix(0,nbins*nbins,3) #one-round hist
for (i in 1:nbins) {
   for (j in 1:nbins) {
         analysis[(i-1) * nbins + j,1] = cuts[i]
         analysis[(i-1) * nbins + j,2] = cuts[j]
   }
}
```

```
for (i in 1:length(gx)) {
   j = as.scalar(gx[i])
   k = as.scalar(gy[i])
   analysis[(j-1) * nbins + k,3] = (analysis[(j-1) * nbins + k,3] +
                                                  b[i]) - (-sign(h[j,k])*h[j,k])
   h[j,k] = h[j,k] + b[i]
}
```

A.2.3 Output Writer

```
nbins = 64
h = matrix(0,nbins,nbins)
```

APPENDIX B Glossary

AoS	Array of Structs
API	Application Programming Interface
CPU	Central Processing Unit
FFI	Foreign Function Interface
FUSE	Filesystem in Userspace
GB	Gigabyte
GCC	GNU Compiler Collection
GSL	GNU Scientific Library
HDF5	Hierarchical Data Format, Version 5
HDFS	Hadoop Filesystem
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
IPC	Inter-Procedural Communication
IR	Intermediate Representation (as in compilers)
JAR	Java Archive
JNI	Java Native Interface
JNR	Java Native Runtime
JVM	Java Virtual Machine
MPI	Message Passing Interface
NetCDF	Network Common Data Format
NFS	Network Filesystem
OS	Operating System
POSIX	Portable Operating System Interface

RAM	Random Access Memory
RDD	Resilient Distributed Dataset (Spark data structure)
RPC	Remote Procedure Call
SPMD	Single Process, Multiple Data (parallel programming paradigm)
SQL	Structured Query Language