Specializing Data Access in a Distributed File System (Generative Pearl)

Pratyush Das
Purdue University
West Lafayette, USA
das160@purdue.edu

Anxhelo Xhebraj
Purdue University
West Lafayette, USA
axhebraj@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, USA
tiark@purdue.edu

Abstract

We propose DDLoader, a system that embeds information about data partitioning and data distribution in distributed file systems using a metaprogramming framework. We demonstrate that this technique has practical benefits for building applications that interact with distributed file system applications. By using metaprogramming, we bring the traditional benefits of staging, such as partial evaluation to the domain of distributed file system access. Furthermore, this approach also fuses the data access code with the computation code, allowing end-to-end optimization. We test our framework on a real-world use case and show that this approach allows users to generate code that performs faster than traditional data loading by up to 12.56x on a single thread and even more when parallelized.

CCS Concepts: • **Software and its engineering → Source code generation**.

Keywords: Staging, Generative Programming, Distributed File Systems

ACM Reference Format:
Pratyush Das, Anxhelo Xhebraj, and Tiark Rompf. 2024. Specializing Data Access in a Distributed File System (Generative Pearl). In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3689484.3690736

1 Introduction

Modern distributed file systems partition and replicate data amongst a cluster of participating nodes to enable processing of larger-than-memory datasets and to provide fault tolerance. Typically, these systems maintain designated coordinator nodes that track the state of the cluster and route user operations to the nodes holding the requested data.

One way to improve performance in this scenario is to provide the client with the cluster topology and data distribution information so that they can bypass the designated coordinator and directly access the data. Loading the files from the underlying file system instead of having to go through high level APIs can lead to significantly better performance. While these might cause concerns due to missing added benefits provided by these high level APIs such as fault tolerance, this is useful for systems where fault tolerance is infrequent, and in the rare case that it does happen, the computation can be restarted.

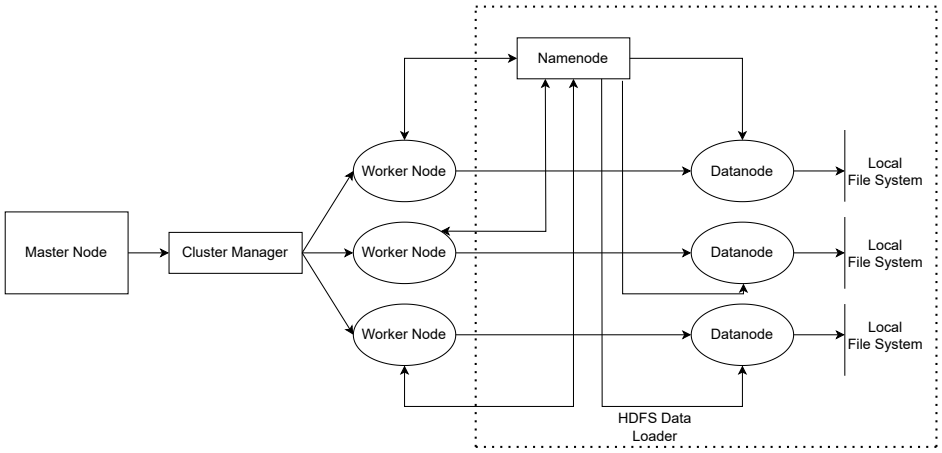
In this paper, we propose a system, DDLoader, that embeds the information about the data distribution into the language using the Lightweight Modular Staging[17] metaprogramming framework for Scala. Multi-stage programming techniques allow developers to build applications while delegating the task of generating low-level file loading details to the generated code. This hides the implementation details of the underlying distributed file system while retaining the performance of handwritten examples. Further, metaprogramming erases the distinction between the data loading and data processing phases of the applications typically built over distributed file systems, allowing whole program optimization, which was not possible previously.

DDLoader presents a Scala frontend to the user while generating low-level C code that bypasses a distributed file system's API abstractions to load data directly from the native file system. This abstracts the distributed file system information away from the user while retaining the performance of handwritten examples designed to operate over the file optimally. Even though DDLoader is implemented in Scala and LMS, the ideas can be generalized to other metaprogramming frameworks in both Scala (Squid [13, 14], Scala 3 [23]) and other languages (BuildIt [4], MetaOCaml [10]).

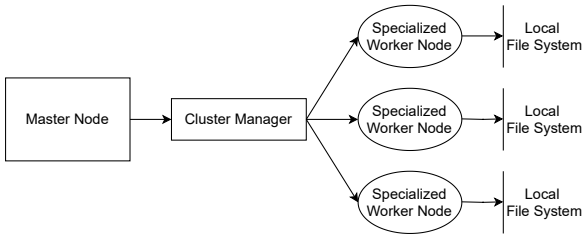
In traditional systems like Apache Spark that can load files from a non-native file system such as the Hadoop Distributed File System (HDFS) [21], the data loading and computation are distinct and independent parts of the system architecture (Figure 1a). DDLoader can query the file system information and generates a unified executable that removes the boundary between the workers and the Datanodes (Figure 1b); thereby loading files faster than an overarching system that uses Hadoop's API abstractions. HDFS' file system itself handles part of fault tolerance by replicating a block on different

GPCE '24, October 21–22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1211-1/24/10
https://doi.org/10.1145/3689484.3690736



(a) Architecture of Apache Spark loading files from HDFS



(b) Apache Spark with a specialized backend to load files from HDFS

Figure 1. Example of a class of systems which could benefit from techniques discussed in this paper

nodes, and only the paths to blocks on live nodes would be returned by DDLoader.

This paper makes the following contributions:

- We discuss the implementation of our new system, DDLoader, that is aware of HDFS' architecture at staging time and utilizes multi-stage programming techniques to embed the information of the underlying distributed file system in the Scala language using LMS (Section 3).
- We demonstrate that DDLoader can operate on files up to 12.56x faster compared to the available HDFS APIs on a single core and more when parallelized (Section 4).

Systems like Flare [9] have explored using metaprogramming to accelerate data processing, but such systems still need to load data through default abstractions such as ones provided by HDFS. By accelerating the data access layer using a system such as DDLoader, this abstraction boundary goes away, yielding potential performance benefits.

2 Background

2.1 Hadoop Distributed File System

HDFS is a distributed file system that is designed to store extremely large datasets. In an HDFS cluster, there is a single Namenode and multiple Datanodes (Figure 1a). The Namenode contains metadata that keeps track of how the data is split and stored in blocks across the Datanodes. The Namenode receives instructions from user processes and uses them to make decisions about allocating read and write operations to specific Datanodes. The Namenode itself does not operate on the data and instead instructs the Datanodes to move the data if required and perform all of the computation.

HDFS Architecture: Let us assume that a user Alice has a 100 TB logical file and wants to store it on a HDFS cluster of 20 machines – which in a typical case, would imply 20 Datanodes, each corresponding to a particular machine. A HDFS cluster comprises a single Namenode which communicates with multiple Datanodes (Figure 1a). The 100 TB file would be split into several blocks and stored across the Datanodes (the sizes of these blocks can be configured by Alice). Each block would in turn be replicated on multiple Datanodes (the replication factor is configured by Alice) to provide the reliability of the cluster. HDFS' block placement policy prioritizes the read bandwidth when deciding which Datanode to replicate and store a block on.

The Namenode stores a snapshot of the current state of the cluster in a file called the FSImage and records all modifications made to the cluster in an edit log, which is periodically synced into the FSImage. The Namenode and the Datanodes communicate with each other using heartbeats (which are a report of the current status of the Datanode) to optimize file storage such as ensuring that the Datanodes have a balanced disk usage. When Alice wants to read or write a file into their cluster, the HDFS client they are accessing the cluster from would first query the Namenode for the available Datanodes and then depending on the API that Alice decides to use, either load the file parallelly on each Datanode in the case of MapReduce or move the data from each Datanode to the HDFS Client when using Hadoop's general API (Hadoop can transparently orchestrate network transfer among nodes in the cluster).

Loading files from HDFS: All data stored in HDFS is stored in the same serialization format as when it was moved to HDFS on the Datanode's native file system as blocks. Each block name is prefixed by a number that denotes the sequence of blocks that make up a particular file to ensure that a file is always read in the correct order. Hadoop has several publicly available APIs that allow loading files from HDFS, in Java (Figure 2a), C (Figure 2b) and Shell (Figure 2c). It is important to note that the Java API is the canonical API since HDFS (and the entirety of the Hadoop ecosystem) is implemented in Java, and the other APIs call the Java API internally – C by using the JNI and the Shell API being Java functions that are executed using a bundled binary executable. The performance of the Java API and the C API is compared in Section 4 (the shell API is significantly slower for our chosen operations).

2.2 Lightweight Modular Staging

Lightweight modular staging (LMS) [17] is a metaprogramming framework written in Scala. The user has control over the generated code by annotating the types, `T` for when the value is known at compile time and `Rep[T]` for when the value has to be computed at runtime.

For example, in the following example, the user has annotated the type of `b` as `Rep[Int]` and `n` as `Int`:

```
def power(b: Rep[Int], n: Int): Rep[Int] =
  if (n == 0) 1
  else b * power(b, n-1)
```

Since `n`'s value is known at compile time, the generated code will be specialized for the value of `n`.

```
int power5(int b) {
  return b * b * b * b * b
}
```

Using LMS provides several advantages such as allowing the user written code to be strongly typed as well as being able to specialize expressive Scala code to generate performant low level code [1, 15, 16, 18, 19, 22].

3 DDLoader

DDLoader is a system that utilizes LMS's multi-staging compilation to generate specialized code that eliminates HDFS abstractions to load data directly from the native file system. Since DDLoader contains information both about the file system architecture as well as the specific computation, it is able to make optimizations on the generated unified executable.

3.1 Querying the file information

Hadoop provides a command (`fsck`) to query the health of the disks where data is stored. This command generates information about the names of the blocks that make up a particular file, the IP address of the physical machine where a particular block is located, the replication factor for a particular file and a part of the path where the file is stored on the physical machine's underlying native file system, relative to the configured HDFS block storage directory on each Datanode. The path of the block storage directory that is not available in the `fsck` output can be retrieved by querying the value of the `dfs.datanode.data.dir` configuration property of the Datanode. Once the full path of each block is available, the blocks can be loaded independently without having to use the HDFS data loading API (Figure 1). Since the `fsck` command generates the IPs of all the machines that a block is replicated across, it is possible to choose the node which would load and operate over a particular block, thus splitting the computation across multiple nodes.

3.2 Interface

To write an operation, the user writes a class that extends our DDLoader trait which is defined as -

```
trait DDLoader ... {
  def HDFSExec(paths: Rep[Array[String]],
       readFunc:(Rep[Int], Rep[LongArray[Char]],
                 Rep[Long]) => RepArray[Char]
               ): Unit
}
```

Here, `paths` is an automatically filled (from first-stage compilation) array where each element is a path to the ith block that makes up the file stored over HDFS. `readFunc` can be either `mmapFile` or `readFile`, which are both defined in `DDLoader.LMSMore`. The former uses Linux's `mmap()` command to read the file, whereas the latter uses Linux's `read()` command.

The user also implements their own `HDFSExec` function which satisfies the signature of the abstract `HDFSExec`, inside some user defined class `UserClass`.

Even though the provided API allows files to be loaded using both `mmap()` and `read()` (neither of the two is optimal in all cases [6]), our evaluations in Section 4 use `mmap()`, since it yielded better performance for the operations compared in this paper. They then create an object of DDLoader's

```java
Configuration conf = new Configuration();
Path inFile = new Path(args[0]);
FSDataInputStream in = fs.open(inFile);
byte[] buffer = new byte[size];
in.readFully(0, buffer, 0, size);
for (i=0; i<size; i++) {
    if (buffer[i] != 32) { count++; }
}
```

(a) Java source code that loads files from HDFS

```c
hdfsFS con = hdfsConnect("127.0.0.1",9000);
char *inFile = argv[1];
hdfsFile op = hdfsOpenFile(con,inFile, 0, 0, 0, 0);
char *buffer = calloc(size, sizeof(char));
hdfsPread(con,op,0,buffer,size);
for (int i = 0; i < size; i++) {
    if (buffer[i] != ' ') { count++; }
}
```

(b) C source code that loads files from HDFS

```bash
hdfs dfs -cat $filepath | tr -d '[:blank:]' | wc -m
```

(c) Bash source code that loads files from HDFS

Figure 2. Example programs that load files from HDFS and count the number of non-whitespace characters

```
@virtualize
class UserClass extends DDLoader {
  override def HDFSExec(...) {
    // Operation code here
  }
}

object UserObject {
  def main(args: Array[String]): Unit = {
    val ops = new UserClass()
    val loadFile  = ...// File to load from HDFS
    val writeFile = ...// C file to generate
    val driver = new DDLDriver(ops, loadFile,
                               mmapFile) {}
    driver.emitMyCode(writeFile)
  }
}
```

Listing 1. DDLoader example

driver called `DDLDriver` and pass the relevant arguments to its constructor. To generate code, call `emitMyCode` on the driver object (Listing 1).

The users do not need to know any additional information when writing a program over an HDFS file using DDLoader, as compared to using one of the official Hadoop APIs. In Listing 2, we demonstrate the body of the `HDFSExec` function for a simple whitespace count program written in DDLoader. As we can see, the program looks very similar to the Java or C Hadoop API implementations in Figure 2.

3.3 Specializations

The benefit of a system such as DDLoader is that it can be easily integrated into other systems that want to load files from HDFS. DDLoader provides a high level API that does not require any more programming effort than using the Hadoop API, but automatically is able to apply a suite

```
val buf = NewLongArray[Char](getBlockLen()+1,
                             Some(0))
var count = 0L
for (i <- 0 until paths.length) {
  val block_num = open(paths(i))
  val size = filelen(block_num)
  val fpointer = readFunc(block_num, buf, size)
  for (j <- 0 until fpointer.length) {
    if (fpointer(j) == ' ') { count = count+1L }
  }
  close(block_num)
}
```

Listing 2. HDFSExec for whitespace count in DDLoader

of optimizations provided by the LMS framework when it generates the code.

Static Data: Code generation can happen either at run time (dynamic) or compile time (static). Multistage programming (in our case, using LMS) allows a combination of the two where DDLoader is able to query Hadoop's Namenode and retrieve the paths of the blocks which make up a file stored in HDFS at stage 1 compile time (scala runtime that generates native code), thereby effectively having zero cost for these utilities.

In Listing 3, we can see that in the generated C code for a simple whitespace count program, the HDFS abstraction layer is no longer present (with no calls to the HDFS API), and the paths to the individual blocks that make up a file are hardcoded into the generated C code.

Dead Code Elimination: LMS allows explicit tracking of effects, by allowing users to specify an operation as being pure or effectful, which is made possible due to its graph intermediate representation (Fig. 10 in Bracevac et al. [3]). For example, if the user opens all the files in an HDFS directory but only operates on specific files in *Stage 1*, the generated code would only open the relevant files instead

```
void Snippet(int x0) {
  char** x1 = (char**)malloc(79*sizeof(char*));
  x1[0] = "path_to_block_0";
  x1[1] = "path_to_block_1";
  ...
  x1[77] = "path_to_block_77";
  x1[78] = "path_to_block_78";
  long x3 = 0L;
  int x4 = 0;
  while (x4 != 79) {
    int x5 = open(x1[x4], 0);
    long x6 = fsize(x5);
    char* x7 = mmap(0, x6, PROT_READ |
    PROT_WRITE, MAP_FILE | MAP_PRIVATE, x5, 0);
    int x8 = (int)x6;
    int x9 = 0;
    while (x9 != x8) {
      if (x7[(long)x9] == ' ') x3 = x3 + 1L;
      x9 = x9 + 1;
    }
    close(x5);
    x4 = x4 + 1;
  }
  printf("%ld\n", x3);
  free(x1);
}
```

Listing 3. Generated C code for a whitespace count program

of all of them. Since the interface provided by DDLoader to the user provides an array containing the paths to the blocks that make up a file stored in HDFS (Listing 1), this is also extensible at a finer granularity and the user can decide to only operate over some of the blocks, instead of the entire file, and DDLoader would only generate code to `open` the relevant blocks. Even though `open` is usually an operation with side effects, LMS allows `open` to be modelled as a function that returns a handle, that if unused, would not be propagated to the generated code.

Parallelization: Since DDLoader is aware of the location of blocks of the file through the `paths` parameter, the user can decide to use `MPI` and write functions to operate over the different blocks in parallel before aggregating information from all the relevant computations if required. This often yields significant performance benefits since data loading is easily parallelizable. The parallelization is defined by the user, and DDLoader provides a set of MPI APIs to make this easier. This type of optimization is more difficult to make when the API provided to the user only allows operations over a single large file.

4 Evaluation

The paper demonstrates the speedups that DDLoader can get by using a staging compiler (LMS) and by bypassing the Hadoop API, over existing methods of loading files from HDFS.

We have implemented different operations over files stored on HDFS with increasing difficulty and have used this to test different aspects of our code generation.

- Whitespace count: computes a single `Long`.
- Character Frequency: computes a fixed-size array.
- Word count: computes a variable length `Map[String => Long]`. A canonical MapReduce[7] example.

Although there is minimal novelty in implementing the above operations, in the context of a metaprogramming framework, the efficient loading of the file is entangled with the computation. Since it is difficult to decouple the computation from the data loading when demonstrating the capabilities of our system, we decided to include the computation in the benchmarks. We have chosen computations that are relatively simple to ensure that the computation time taken for processing does not overshadow the data loading time. The complexity of our experiments builds up to word count, which might be a simple example in distributed settings, but demonstrates techniques that are useful in real distributed programs, such as fusing hashmaps across nodes.

The benchmarks have been executed on a NUMA machine featuring 4 sockets (comprising 8 NUMA nodes), each with 24 Intel(R) Platinum 8168 cores and 750GB RAM per socket. Both the HDFS files and the native files are stored on a mdraid RAID 0 array made up of 6 NVME disks. Even though HDFS is usually used in a distributed setting, our setup contains several NUMA nodes where there is a higher penalty to access remote memory regions, which is representative of the fact that DDLoader can scale to real distributed system workloads. The files stored in HDFS that have been used for these evaluations have a mostly uniform distribution of words. While benchmarking our programs, we start the timer after the call to `MPI_Init()` to minimize MPI setup cost.

In Figure 3, we compare the time taken to perform whitespace count, character frequency and word count by Java and C code (using the Hadoop API) and DDLoader. The plots use the Java code which is the canonical Hadoop implementation as a baseline, and demonstrate the speedup of the alternate implementations. The C code (using Hadoop's C API) is usually slightly faster, while DDLoader's performance is significantly faster than the alternatives. This is because DDLoader bypasses Hadoop API overheads, which the C and Java benchmarks incur, by moving some Hadoop operations to the first stage compile time and generating a specialized executable.

In our three sample programs (whitespace count, character frequency and word count) implemented using DDLoader, we have implemented distributed versions of our operations using MPI. Although the benchmarks have been run on a single node machine, thereby making the operations "parallel" instead of "distributed", the implementation has been

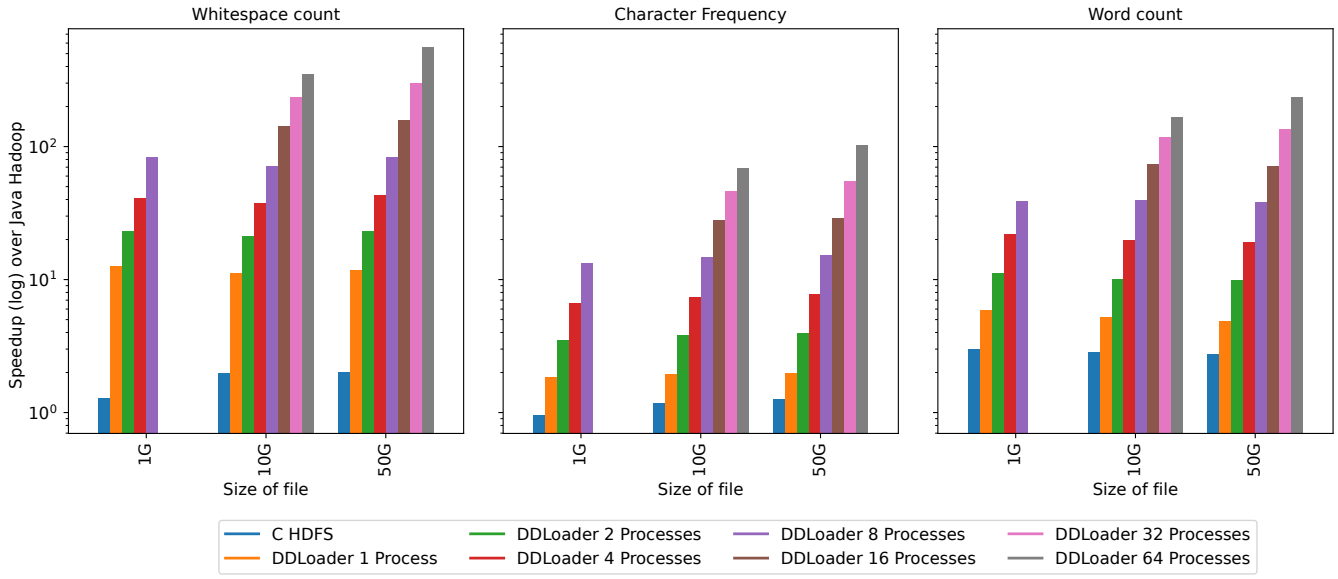


Figure 3. Different operations over files stored in HDFS - Hadoop API vs DDLoader

Table 1. Computing the number of white spaces in a file stored in HDFS

File size	Java Hadoop	C Hadoop	DDLoader 1 proc	DDLoader 2 procs	DDLoader 4 procs	DDLoader 8 procs	DDLoader 16 procs	DDLoader 32 procs	DDLoader 64 procs
1 GB	4.63s	3.6s (1.29x)	0.37s (12.56x)	0.2s (23.15x)	0.11s (41.04x)	0.06s (82.93x)	x	x	x
10 GB	40.02s	20.21s (1.98x)	3.54s (11.29x)	1.87s (21.4x)	1.06s (37.7x)	0.56s (71.28x)	0.28s (142.47x)	0.17s (236.32x)	0.11s (352.41x)
50 GB	206.35s	101.75s (2.03x)	17.5s (11.79x)	8.95s (23.05x)	4.73s (43.59x)	2.49s (83.02x)	1.31s (157.56x)	0.68s (301.45x)	0.37s (559.24x)

Table 2. Computing the frequency of each character in a file stored in HDFS

File size	Java Hadoop	C Hadoop	DDLoader 1 proc	DDLoader 2 procs	DDLoader 4 procs	DDLoader 8 procs	DDLoader 16 procs	DDLoader 32 procs	DDLoader 64 procs
1 GB	5.89s	6.15s (0.96x)	3.18s (1.86x)	1.67s (3.53x)	0.89s (6.62x)	0.44s (13.28x)	x	x	x
10 GB	54.87s	46.41s (1.18x)	28.29s (1.94x)	14.28s (3.84x)	7.39s (7.42x)	3.73s (14.71x)	1.97s (27.9x)	1.19s (46.08x)	0.8s (68.98x)
50 GB	283.61s	223.15s (1.27x)	141.81s (2.0x)	72.0s (3.94x)	36.46s (7.78x)	18.38s (15.43x)	9.85s (28.8x)	5.17s (54.82x)	2.78s (101.94x)

Table 3. Computing the frequency of each word in a file stored in HDFS

File size	Java Hadoop	C Hadoop	DDLoader 1 proc	DDLoader 2 procs	DDLoader 4 procs	DDLoader 8 procs	DDLoader 16 procs	DDLoader 32 procs	DDLoader 64 procs
1 GB	34.99s	11.67s (3.0x)	5.92s (5.91x)	3.13s (11.17x)	1.58s (22.13x)	0.9s (38.78x)	x	x	x
10 GB	284.45s	100.42s (2.83x)	54.7s (5.2x)	28.12s (10.12x)	14.25s (19.96x)	7.22s (39.42x)	3.87s (73.48x)	2.42s (117.46x)	1.71s (166.23x)
50 GB	1339.45s	489.58s (2.74x)	272.47s (4.92x)	135.5s (9.88x)	69.43s (19.29x)	35.24s (38.01x)	18.72s (71.56x)	9.86s (135.87x)	5.65s (237.02x)

designed with distribution as the goal and therefore does not make assumptions such as all data being available locally. This is atypical for benchmarks over a distributed file system, but in this case, the evaluations are meant to demonstrate that it is possible to provide high level abstractions while still being able to generate efficient code specialized to the underlying file system infrastructure, instead of improving communication between different nodes in a distributed setting. The whitespace count and the character frequency distributed versions have been implemented using the `MPI_Reduce` intrinsic, whereas the more complicated distributed version of word count required `MPI_Send`, `MPI_Recv`, `MPI_Allgather` and `MPI_Gatherv`. We decided to use these particular APIs since they seem to represent how a user would typically implement these operations while still expecting performance benefits.

In Tables 1, 2 and 3, we can see that our implementation scales with the addition of more processes for files as large as 50GB. DDLoader gets the maximum speedup over the Hadoop API implementations for the Whitespace count operation (Table 1), since DDLoader's generated C code was vectorized by gcc. However, as we found out, using `MPI_Reduce`

on a variable updated in the *to be* vectorized loop can prevent gcc from vectorizing the loop, and therefore care must be taken when using MPI intrinsics – in this case, we copied the variable into a new variable that we passed to `MPI_Reduce` to enable vectorization of the loop.

The parallelization is at the granularity of the file blocks. Under the default HDFS configuration, a 10 GB file is split into 79 blocks. If we assign 64 processes to perform a computation over the 10 GB file, 15 processes will perform the operation over 2 blocks, while the remaining 49 processes will perform the oepration over 2 blocks. This implies that performance will not scale fully linearly if the number of blocks is not exactly divisible by the number of processes. Parallelization beyond 8 processes for a 1GB file was not possible since a 1GB file is split into 8 blocks when stored in HDFS under the default configuration.

5 Limitations

The system described in the paper is a minimal implementation to demonstrate how high level APIs over file system abstractions can be provided as a programming model to the user, while using a metaprogramming framework to generate appropriate low level system calls.

As such, DDLoader has several limitations, such as no built in fault tolerance. However, the system is still useful in cases where resilience is implied or handled by a different layer of the system, for eg. batch jobs where the job can be restarted based on a checkpoint. As future work, it is possible to extend DDLoader, where the runtime dispatches to precompiled C functions. If a particular node is down, the Namenode can be queried again for the new location of the chunk, and the appropriate functions can be re-compiled and then dispatched to.

Similarly, logic for parallel computation and co-ordination over multiple nodes using MPI currently has to be handled by the user provided code, instead of being automatically inferred by DDLoader. While we have re-implemented a part of the HDFS data loading infrastructure for DDLoader, it is possible to re-implement more advanced HDFS features such as encryption zones and distributed tracing as well. This can either be done by implementing the HDFS algorithms ourselves in LMS from scratch, finding corresponding libraries in C, if they exist, that our generated program could link with, or by calling into relevant portions of the open source HDFS Java implementation from our generated C code via JNI (which might have performance penalties due to having to go through JNI, but would require lesser re-implementation effort).

6 Related Work

The traditional technique for processing HDFS data is MapReduce [7]. However, MapReduce has limitations [8, 11, 20], such as certain programs being difficult to express and a generic programming model like DDLoader might be a more suitable alternative for HDFS data.

Although we decided to use LMS to implement DDLoader, the ideas can be generalized to multiple other metaprogramming systems in Scala (Squid [13, 14], Scala 3 [23]), C++ (BuildIt [4]), OCaml (MetaOCaml [10]) and others. Scala 3 provides better ecosystem support since the metaprogramming framework is part of the language itself, however using LMS allows DDLoader to generate C code that directly invokes system calls. Compared to the other metaprogramming systems, LMS allows more fine grained tracking of effects by allowing users to define operations as pure or effectful nodes in a dependency graph as it uses a graph intermediate representation [3, 5]. This can be replicated in a tree based intermediate representation, but it would imply more implementation effort in the form of writing an analysis (for e.g. dataflow analysis) over the intermediate representation. Scala 3, Squid and MetaOCaml rely on term level annotations to make the distinction between stages explicit, while LMS only requires type level annotations which allows the user to write the same code for both staged and unstaged contexts.

Jet [1] is a DSL that uses LMS to generate code to perform MapReduce-like operations in Apache Spark [26] and Apache Hadoop [2]. However, Jet and DDLoader have different concerns – Jet accelerates the operations while DDLoader performs optimizations to load data from HDFS faster. Additionally, Jet is only able to perform a certain set of operations whereas DDLoader allows users to write any programs that can be expressed using LMS. DiSh [12] uses similar techniques as the ones proposed in this paper to query the HDFS file topology, but whereas DDLoader focuses on accelerating the data loading, DiSh implements efficient parallelization of shell scripts that operate over files stored in HDFS.

Apache Spark [26] is a big-data processing system that loads and operates over files stored in HDFS using Hadoop's API. Future work involves integrating DDLoader's multi-stage code generation into Spark's backend for faster file loading. Flare [9], an accelerator backend for Apache Spark using LMS, demonstrates the use of multi-stage programming to speed up computation for small to medium-sized workloads. DDLoader has potential to extend Flare and improve performance for large datasets across a cluster.

7 Conclusion

The paper demonstrates a system called DDLoader to load files from HDFS faster than the provided Hadoop APIs. In particular, using multi-stage programming techniques as implemented in DDLoader allows compilers to optimize a unified code representation that contains information about loading native files as well as the operations to be performed on them.

This paper focuses on a distributed file system like HDFS to prototype our system, since the abstractions are all at the

user level. However, DDLoader could be extended to any distributed file system such as Ceph [25] or local file system such as FuseFS [24].

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We would also like to thank Supun Abeysinghe, Patrick Lafontaine, Anmol Sahoo, Kirshanthan Sundararajah and Guannan Wei for their comments on earlier drafts. This work was supported in part by DOE award DE-SC0018050.

References

[1] Stefan Ackermann, Vojin Jovanović, Martin Odersky, and Tiark Rompf. 2012. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-to-end Management of Big Data*. https://infoscience.epfl.ch/handle/20.500.14299/85985

[2] Apache Software Foundation. 2010. Hadoop. https://hadoop.apache.org

[3] Oliver Bracevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 400–430. https://doi.org/10.1145/3622813

[4] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

[5] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, Michael D. Ernst (Ed.). ACM, 35–49. https://doi.org/10.1145/202529.202534

[6] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p13-crotty.pdf

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html

[8] Christos Doulkeridis and Kjetil Nørvåg. 2014. A survey of large-scale analytical query processing in MapReduce. *VLDB J.* 23, 3 (2014), 355–380. https://doi.org/10.1007/S00778-013-0319-9

[9] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

[10] Oleg Kiselyov. 2023. MetaOCaml Theory and Implementation. https://doi.org/10.48550/ARXIV.2309.08207 arXiv:2309.08207

[11] Jimmy Lin and Chris Dyer. 2010. Data-Intensive Text Processing with MapReduce. (2010). https://doi.org/10.2200/S00274ED1V01Y201006HLT007

[12] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. 2023. DiSh: Dynamic Shell-Script Distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 341–356. https://www.usenix.org/conference/nsdi23/presentation/mustafa

[13] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted staged rewriting: a practical approach to library-defined optimizations. (2017), 131–145. https://doi.org/10.1145/3136040.3136043

[14] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, 56–66. https://doi.org/10.1145/3136000.3136005

[15] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 2–9. https://doi.org/10.1145/2784731.2784760

[16] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 238–261. https://doi.org/10.4230/LIPICS.SNAPL.2015.238

[17] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[18] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510. https://doi.org/10.1145/2429069.2429128

[19] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 41–52. https://doi.org/10.1145/2594291.2594316

[20] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. 2013. The Family of Mapreduce and Large-Scale Data Processing Systems. *ACM Comput. Surv.* 46, 1, Article 11 (jul 2013), 44 pages. https://doi.org/10.1145/2522968.2522979

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, Mohammed G. Khatib, Xubin He, and Michael Factor (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[22] Alen Stojanov, Tiark Rompf, and Markus Püschel. 2019. A stage-polymorphic IR for compiling MATLAB-style dynamic tensor expressions. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm (Eds.). ACM, 34–47. https://doi.org/10.1145/3357765.3359514

[23] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. https://doi.org/10.1145/3278122.3278139

[24] Miklos Szeredi. 2010. Fuse: Filesystem in Userspace. https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[25] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 307–320. http://www.usenix.org/events/osdi06/tech/weil.html

[26] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

Received 2024-06-18; accepted 2024-08-15