D4.3 Improved DSL Runtime Prototype and Overview

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.0

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

In this Deliverable, we describe the status of the DAPHNE runtime system and its prototype v2 implementation. Specifically, we overview the current system architecture and continue with the implementation and integration updates that took place within this reporting period. We can categorize progress made along the following axes: a) Advances in runtime I/O; b) updates in runtime communication and c) updates inside the execution engine itself. We provide access to the DAPHNE Runtime v2 prototype, publicly available in the DAPHNE development repository. We conclude this report by including examples on how to execute sample DSL algorithms with various parameters and provide a large set of benchmarking results of the current version over the VEGA supercomputer.

D4.3 Improved DSL Runtime Prototype and Overview			
WP4 – DSL Runtime and Integration			
Type of document	R	Version	1.0
Dissemination level	PU		
Lead partner	ICCS		
Author(s)	Dimitrios Tsoumakos (ICCS) Stratos Psomadakis (ICCS), Constantinos Bitsakos (ICCS), Aristotelis Vontzalidis (ICCS), Mark Dokter (KNOW), Patrick Damme (TUB)		
Reviewer(s)	Jonas H. Müller Korndörfer (UNIBAS), Philippe Bonnet (ITU)		

Revision History

Version	Item	Comment	Author / Reviewer
v0.1	Initial outline	16/10/2023	Dimitrios Tsoumakos
v0.2	Updated outline	25/10/2023	Dimitrios Tsoumakos
v0.3	Introduction content	3/11/2023	Dimitrios Tsoumakos
v0.4	Content on updated runtime design	6/11/2023	Aristotelis Vontzalidis, Stratos Psomadakis
v0.5	Content on HDFS integration and Eigen kernel	8/11/2023	Constantinos Bitsakos
v0.6	Experimental evaluation section	9/11/2023	Aristotelis Vontzalidis
v0.7	Updated experimental results and commentary	10/11/2023	Aristotelis Vontzalidis, Dimitrios Tsoumakos
v0.8	Integrate more kernel input	11/11/2023	Patrick Damme, Dimitrios Tsoumakos
v0.9	Updated content based on reviews	23/11/2023	Aristotelis Vontzalidis, Constantinos Bitsakos
v1.0	Updated content	26/11/2023	Dimitrios Tsoumakos

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CSV	Comma-seperated Values
CUDA	Compute Unified Device Architecture
DSL	Domain Specific Language
GPU	General Processing Unit
HPC	High Performance Computing
HDFS	Hadoop Distributed File System
IDA	Integrated Data Analysis
I/O	Input / Output
JVM	Java Virtual Machine
ML	Machine learning
MLIR	Multi-Level Intermediate Representation
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communications Library
OpenMP	Open Multi-Processing
RPC	Remote Procedure Call

1 Introduction

Complex end-to-end analysis requirements of modern analytics create a definite trend towards Integrated Data Analysis pipelines (IDAs) where data management, high-performance computing and ML tasks are arbitrarily "mixed-and-matched". Distinctive such use-cases or domains include ML-assisted simulations, exploratory query processing, data cleaning, etc. Nevertheless, the deployment of IDA pipelines is currently hindered by integration challenges among different systems and hardware issues [D+22].

The DAPHNE project[1] is building an open and extensible system infrastructure for such integrated data analysis pipelines. DAPHNE is built on top of MLIR [L+21], allowing seamless integration with existing applications and runtime libraries while also enabling extensibility for specialized data types, hardware-specific compilation chains and custom scheduling algorithms. Its technical contributions are available as open source under the Apache-2.0 license[2].

In this deliverable we present the advances in design and implementation relative to the DAPHNE Runtime that have been achieved during the past year. We share a snapshot of the latest DAPHNE Runtime prototype, detail how users can use it and provide detailed benchmarking results on its performance across important input, resource and configuration dimensions. Previous relative deliverables include the refined DAPHNE system architecture [D+22a] and the DSL Runtime Prototype [V+22]. We also note that this prototype and document are the result of the collaborative work that is performed by consortium partners that participate in WP4.

1.1 Overview of DAPHNE System Architecture

Figure 1 shows the DAPHNE system architecture [D+22a]. In short, users specify their IDA pipelines in the DaphneDSL (a language similar to Julia, PyTorch, or R) or DaphneLib (a high-level Python API that internally compiles to DaphneDSL scripts as well). These scripts are then compiled, by a multi-level compilation chain, into executable runtime plans, which can be executed in a local or distributed environment. DAPHNE is constantly getting updated with diverse tools, e.g., for getting insights into the compiler and runtime.

The multiple compiler passes lower the IR from high-level operations to specialized operator and data levels (C++ Kernels), allowing just-in-time compilation. Optimization passes improve runtime and memory consumption, including common subexpression elimination, constant propagation, and more. This system infrastructure also supports cardinality and sparsity estimation.

The compiler generates an execution plan with calls to C++ host kernels for local, distributed, or accelerator operations. At DAPHNE Runtime, the kernels are executed according to the

[1] https://daphne-eu.eu/
[2] https://github.com/daphne-eu/daphne

provided execution plan. Besides this basic execution mode, DAPHNE adopts state-of-the-art optimizations in the Runtime that include: task-parallel loops and operations, data-parallelism across nodes, devices, and cores and vector-instruction-parallelism where operations can be adaptively fused and seamless integration of heterogeneous computing devices and distributed operations occurs. In Figure 1, we denote the different architecture levels of DAPHNE where the Runtime integrates/participates in with a green line. With respect to the other DAPHNE components (and corresponding work packages – WPs), the runtime system lies in the heart of the architecture, as it leverages the compiler's output (WP3: "DSL Abstractions and Compilation") to execute kernels based on the various levels of the scheduling logic (WP5: "Scheduling and Resource Sharing") on the available hardware resources (WP6: "Computational Storage" and WP7: "Hardware Accelerator Integration").

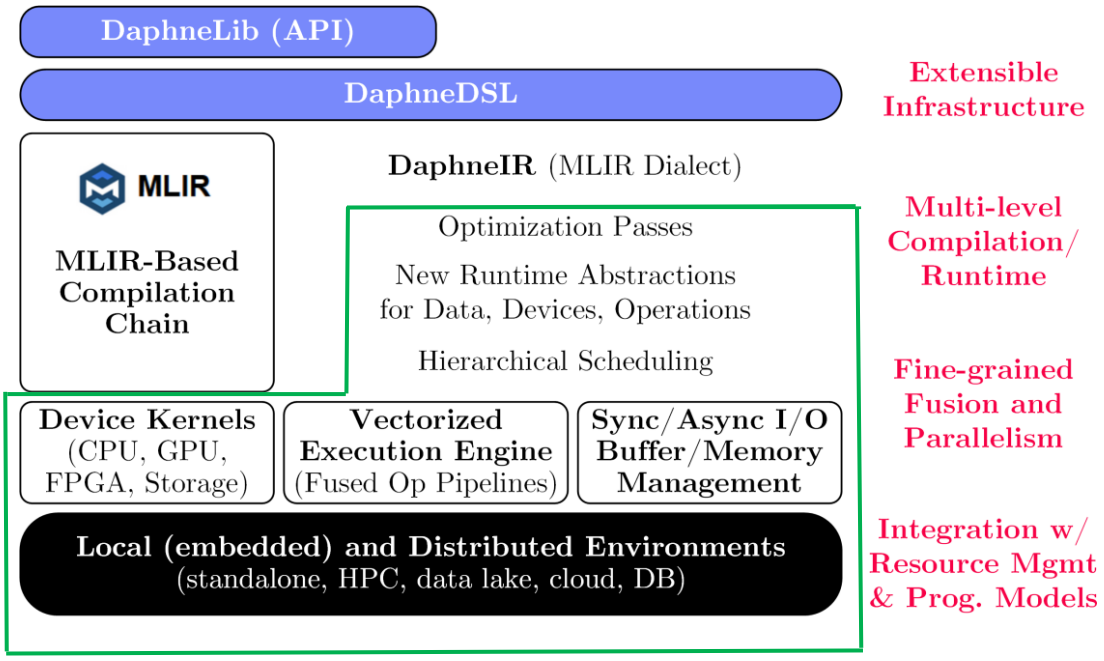


Figure 1: DAPHNE System Architecture and Runtime involvement.

1.2 Runtime Overview

The DAPHNE Runtime system is a crucial component of DAPHNE. It supports the execution of user-defined workflows and operations specified in DaphneDSL or DaphneLib. Below, we identify the different components/integration points that constitute parts of the DAPHNE Runtime efforts together with a brief description for each one:

1.2.1 DAPHNE Runtime

The DAPHNE Runtime is responsible for efficiently executing user-defined IDAs. The runtime system is designed hierarchically: The coordinator receives DaphneDSL user code and generates an execution plan. The compiler determines whether each workload should be executed locally or across multiple worker nodes. This is pictorially shown in Figure 2.

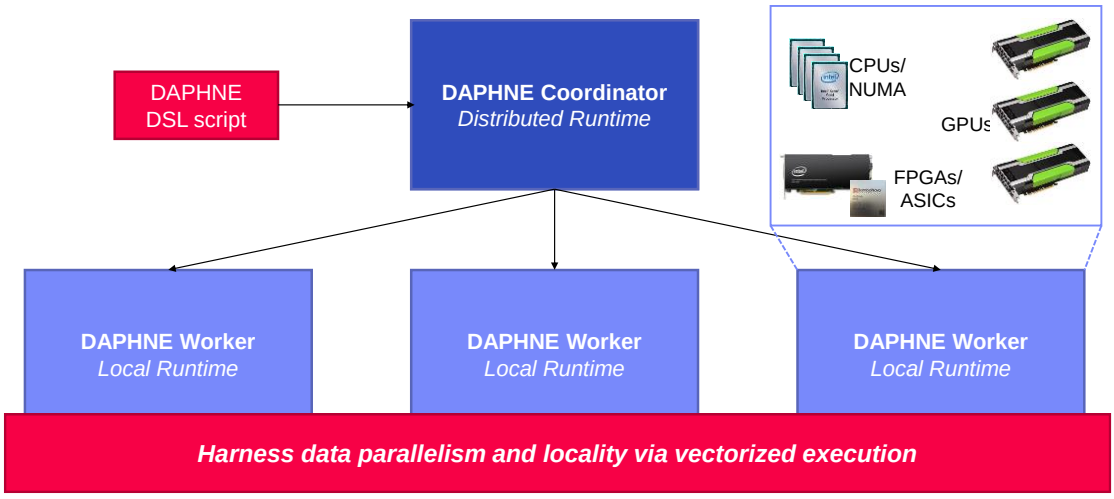


Figure 2: DAPHNE Runtime hierarchical approach.

The *local runtime* is responsible for the execution of complex pipelines in a single compute node. Note that the compute node may consist of multiple heterogeneous resources, e.g., multicore CPUs, GPUs, FPGAs, and computational storage devices. The *distributed runtime system* coordinates the distribution of work among worker nodes and collects the results. Work performed by the compiler determines the code to be executed by each node. This code is sent to the workers in the form of an MLIR snippet, which can be compiled in an architecture-aware manner at the individual workers. The distribution of the data and spawning of distributed jobs is done at the beginning of a fused pipeline.

The "heart" of the runtime is its *vectorized execution engine*, operating as the central means of parallelism for both the local (e.g., multi-threading and multi-device) and the distributed (e.g., multiple worker nodes) runtime. It allows fine-grained operator fusion and parallelism across hardware devices.

In distributed execution mode, the DAPHNE Runtime uses distribution primitives (such as broadcast, all-reduce, ring-reduce, scatter/gather, etc.) to distribute data and code to worker nodes. Instead of CPUs, there are multiple distributed nodes that receive chunks of data and perform computations on them. Each worker locally compiles and executes the generated code through the local runtime system via the vectorized execution engine. This is pictorially described in Figure 3 for the Connected Components algorithm.

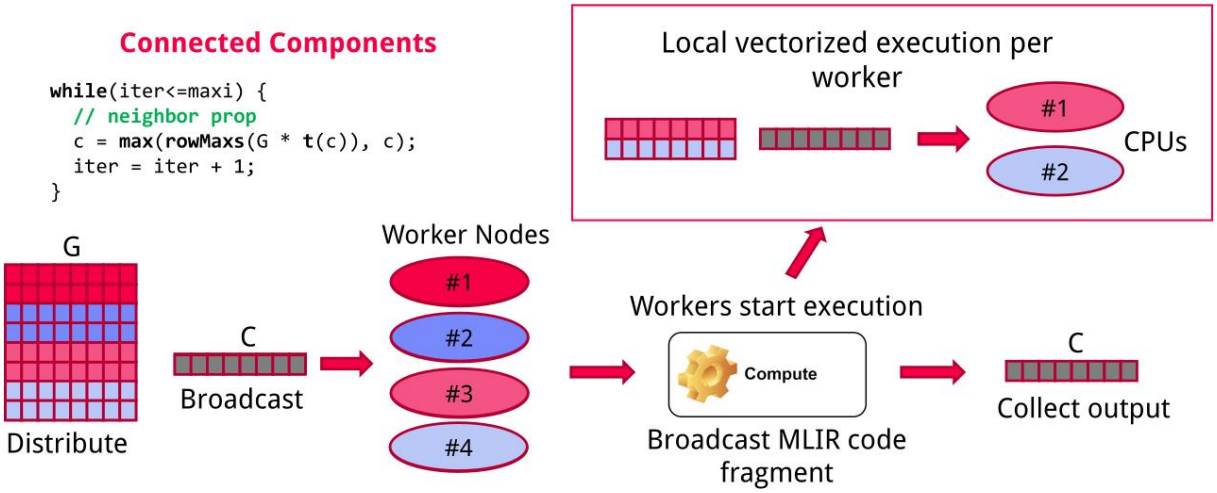


Figure 3: Example of hierarchical and vectorized execution for the Connected Components algorithm.

1.2.2 DAPHNE Kernels

Kernels are the actual code that is executed on a device's hardware, such as a CPU or GPU, and is used to perform a specific operation. In DAPHNE, kernels are written in C++ and are used for local, distributed or accelerator operations. The kernels are specialized for different types of input and output data. In [K21], a summary of the main categories of operations that are supported has been detailed. A full list of available kernels can be found in the project's GitHub page[3].

1.2.3 DAPHNE Runtime Data Structure Support

The Daphne Runtime supports data structures such as matrices (both dense and sparse formats of various extensions) and frames that have a schema and rely on column-oriented storage. In the distributed case, these become distributed collections of tiles and federated matrices/frames. These two abstractions provide a great balance of flexibility and control [K21].

1.2.4 DAPHNE Communication Framework

Communication between cluster nodes is commonly required in multiple DAPHNE operations such as data and code transfer and intra-cluster communication. DAPHNE's extensible architecture allows for easy integration with different communication frameworks, each with different capabilities and performance trade-offs. Currently, DAPHNE Runtime can successfully utilize gRPC (both in synchronous and asynchronous mode) as well as the openMPI library in order to support the required communication.

1.2.5 DAPHNE Runtime I/O

Efficient I/O support for DAPHNE Runtime is of high importance, as most IDA pipelines routinely involve both large and possibly multiple input/output datasets with iterative distributed computations that require large numbers of I/O operations.

Supporting efficient I/O spans different aspects of the runtime:

[3] https://github.com/daphne-eu/daphne/blob/main/src/runtime/local/kernels/kernels.json

Storage backends: DAPHNE Runtime has the ability to utilize different storage backends in order to read and write data from/to. It has been initially deployed with local file system support for each node. During this period, we have opted for integration with the Hadoop Distributed File System (HDFS) [Ha].

(De)Serialization formats: A category of kernels relative to I/O provide support for the DAPHNE user to read and store files in popular formats such as CSV, Arrow and Matrix Market. It is also noteworthy to mention that a DAPHNE-specific file format along with custom (de)serialization support has been devised for more efficient I/O and network communication.

1.3 Document Organization

The remainder of this document is organized as follows: Section 2 describes all the updates performed during the past year, categorized. In Section 3, we first present the artifact for the v2 runtime prototype and how to use it. Then, we detail a set of benchmarking results over different important dimensions that affect the runtime performance. We conclude this document in Section 4.

2 Runtime Design and Implementation Updates

This section describes the updates in the design and implementation of the DAPHNE Runtime system that took place within the last reporting period (i.e., since D4.2 [V+22] in November 2022). Moreover, it provides information regarding the current status of the prototype, where applicable.

2.1 Overview

We can categorize progress made in this period along the following axes: a) Advances in I/O; b) updates in runtime communication and c) updates inside the execution engine itself.

Relative to I/O in DAPHNE Runtime: We have updated the DAPHNE serialization library to support DenseMatrices and CSRMatrices. Three serialization methods are implemented, offering flexibility in preserving and reconstructing DAPHNE objects. This not only improves I/O performance but also broadens support for distributed execution to various data types. We have also commenced an integration of the runtime with HDFS. We are leveraging the *Libhdfs3* library for efficient interfacing with HDFS from C++ code. The workflow involves file upload, serialization, data distribution, information gathering, deserialization, and inter-node write and read. This integration is planned to greatly enhance data locality and establish a scalable distributed computing environment.

Relative to Runtime Communication advances: DAPHNE's communication capabilities are enhanced with the full incorporation of MPI and the introduction of synchronous gRPC alongside asynchronous gRPC. To overcome the limitation of message size in both frameworks (as well other possible future integrations), we have introduced message fragmentation. This enables the system to handle larger data volumes by breaking them into manageable segments, seamlessly facilitated by the serialization library.

Execution Engine Updates: The execution engine underwent several updates, including metadata support for both gRPC and MPI, integration with the Eigen library for eigenvalue and eigenvector calculations, CSR support for PageRank, and CUDA-specific kernels. Moreover, basic monitoring support for the runtime is added through the PAPI library, allowing fine-grained event information extraction.

2.2 I/O in Daphne Runtime

2.2.1 Daphne Serialization

We have implemented a DAPHNE serialization library[4] for `DenseMatrices` and `CSRMatrices` (sparse matrices). This serialization library transforms Daphne objects into byte streams in a fast and efficient manner, leveraging the underlying structure. We have implemented three distinct serialization methods, as described below:

[4] https://github.com/daphne-eu/daphne/blob/main/src/runtime/local/io/DaphneSerializer.h

- The first method is a straightforward approach that serializes the entire object into a byte array. This method provides a complete snapshot of the object's state, making it ideal for situations where the entire object needs to be preserved and reconstructed in its entirety. It simplifies the process by encapsulating the object in a compact byte array.
- The second method introduces the concept of serialization in *chunks*, utilizing C++ iterators to break down the serialized object into smaller, manageable segments, instead of a single compact byte array. This approach allows for the creation of smaller segments (i.e., *chunks*) of the serialized object one at a time, when dealing with exceptionally large objects or cases where creating a byte array in its entirety would lead to excessive memory usage. This method is particularly useful when it is essential to manage memory efficiently. The implementation is based around C++ iterators; the DAPHNE programmer can create Serialization objects that can be iterated via a C++ for loop, producing chunks in each iteration.
- The third method takes serialization in chunks a step further by allowing out-of-order (de)serialization. Each chunk contains sufficient header information, enabling them to be stored or transmitted independently. The header is the same as the one used for the DAPHNE binary format[5], necessary for reconstructing the DAPHNE object, plus an index, which is used to identify each individual chunk. This means that the chunks can be used to reconstruct the object in any order, providing a high degree of flexibility in managing serialized data. This method is particularly valuable in scenarios where chunks may be processed, stored, or transmitted separately and reassembled later as needed.

The DAPHNE serialization library not only enhances I/O performance but also facilitates faster communication for the distributed runtime. This is achieved by eliminating the need to employ *protobuf* semantics for transferring DAPHNE objects.

Furthermore, it enables us to utilize the distributed runtime with a wider range of data types. Up to this point (as described in [V+22]), our support for distributed execution has been limited to DenseMatrices of type double. However, we can now transfer any object type that is supported by the library, from the coordinator to the workers (note that the transfer from workers back to the coordinator is still in development and limited to *DenseMatrix<double>* only).

2.2.2 FileSystem Integration – HDFS

2.2.2.1 HDFS Overview

The Hadoop Distributed File System (HDFS) is a scalable, fault-tolerant file system used by Hadoop applications to efficiently distribute large datasets across multiple nodes. By design, HDFS facilitates data locality, thereby minimizing network congestion and improving system throughput.

[5] https://github.com/daphne-eu/daphne/blob/main/doc/BinaryFormat.md

HDFS is a sophisticated file system framework designed to store large datasets across numerous commodity machines. HDFS follows a master/slave architecture comprising a single *NameNode* for managing the file system metadata and regulating access, and multiple *DataNodes* to handle data storage on individual cluster nodes. Files are split into large blocks, which are grouped at datacenter racks, distributed and replicated among DataNodes, ensuring high fault tolerance. The NameNode orchestrates the block storage, while DataNodes serve read and write requests from the clients. These are pictorially described in Figure 4. This design allows for high throughput and the capability to handle thousands of nodes and petabytes of data, positioning HDFS as a basic "go-to" big data storage platform.

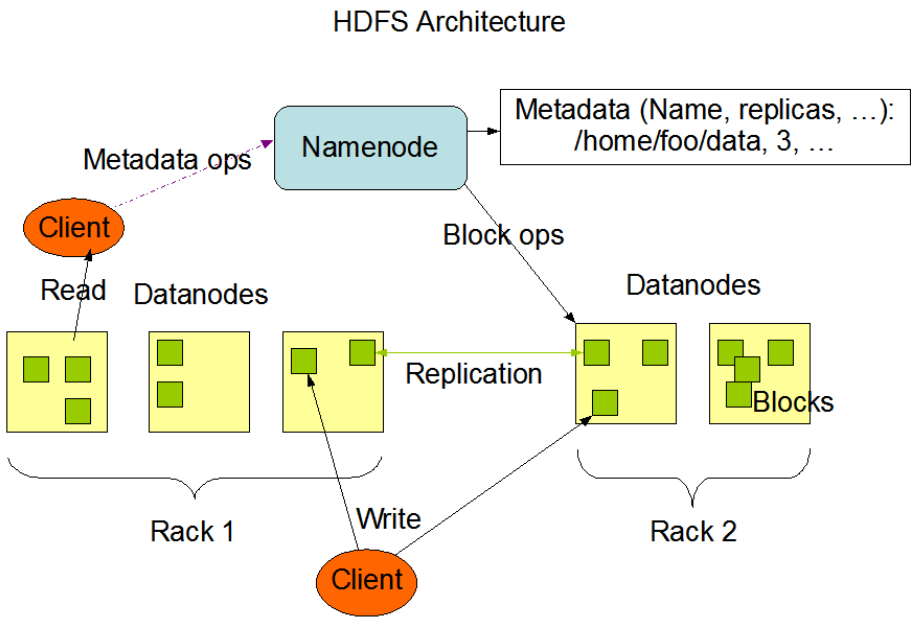


Figure 4: HDFS Architecture (https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html)

2.2.2.2 HDFS APIs

In interfacing with HDFS from C++ code, several libraries are available, each with unique advantages and limitations. The traditional JNI-based C library, *libhdfs*, requires deployment of HDFS JARs on every machine, which leads to a longer startup time. Alternatively, *WebHDFS* provides a REST API for HDFS interaction. However, crafting HTTP requests in C/C++ is relatively cumbersome, and not particularly open to performance optimizations. *libwebhdfs*, a C library designed for WebHDFS requests, provides a way to interact with Hadoop's HDFS using a REST API, avoiding the need for Hadoop Java libraries on each machine. This approach simplifies deployment in large-scale environments by eliminating the requirement to install Java JARs everywhere. Additionally, it bypasses the Java Virtual Machine (JVM) startup time, offering quicker and more efficient access to HDFS. Consequently, libwebhdfs is preferrable in cases where reducing Java overhead and streamlining HDFS access are essential.

The *Libhdfs3* library offers a significant advancement as a native C/C++ HDFS client. It operates independently of the JVM, providing seamless support for non-JVM languages like Python and eliminating the necessity for Hadoop JARs or JVM system libraries. This independence from the Java environment not only streamlines the process but also enhances performance by avoiding JNI overhead.

For this project, Libhdfs3 was selected due to its native implementation that offers a more direct, and thus faster, interaction with HDFS. It streamlines deployment and execution by removing the dependency on Java infrastructure, which can be a substantial advantage in environments where Java is not the primary runtime. Moreover, Libhdfs3 generally provides a more idiomatic C/C++ development experience, which can be beneficial for C/C++ developers. These reasons, coupled with its robustness and efficiency, make Libhdfs3 the preferred choice for our HDFS-related operations.

2.2.2.3 Current integration status

The integration of DAPHNE with HDFS signifies a strategic advancement in managing distributed data processing tasks. In terms of the DAPHNE Runtime, our primary goal is to enhance its performance by taking advantage of data locality. Additionally, we wish to showcase DAPHNE's extensibility by integrating a state-of-the-art distributed file system with it. The use of Libhdfs3 is instrumental in bridging DAPHNE with HDFS, ensuring that the runtime leverages its strengths. The following outlined workflow, not only promotes data locality but also paves the way for a scalable and robust distributed computing environment.

1. File Upload: Users initiate the process by uploading a file to HDFS. This action sets the stage for distributed data handling by our system.
2. Serialization: The DAPHNE coordinator serializes the contents of the file into fixed-size DAPHNE chunks[6]. This serialization is a preparatory step for aligning the data with HDFS's block structure.
3. Data Distribution: Corresponding to the DAPHNE chunks, the data is then sharded into fixed-size HDFS chunks with size instructed by the coordinator. This ensures that the data distribution mirrors our chunking performed in step 2 above. The size of DAPHNE and HDFS chunks is dynamically determined during execution, based on the number of nodes and the size of the original matrix. This adjustment aims to achieve load balancing, ensuring that each node can efficiently deserialize the bytestream objects it receives for performing its computations.
4. Information Gathering: Each worker node, after being prompted by the coordinator, reports back detailing which file blocks are stored in its local disk. This information is crucial for tracking data and scheduling tasks.
5. Deserialization and Matrix Formation: Each node proceeds to deserialize its respective blocks, resulting in the formation of a corresponding DAPHNE data format usable for computation. In this manner, each node reads only the portions of the original matrix stored on its local disk from HDFS, thereby achieving data locality and minimizing intercommunication overhead.
6. Inter-node Write and Read: Each node is capable of writing its output directly back to HDFS. The master node can then read these outputs without the necessity of a collect function, streamlining the data aggregation process.

[6] The term *chunks* refers to its definition from Section 2.2.1 – the Daphne serializer. These are essentially segments of the original matrix represented as bytestream chunks.

7. Task Assignment: The final step (currently under development), will see the coordinator dispatching job details to each node. Each node will be responsible for a specific portion of the initial file, facilitating parallel processing and efficiency.

In the current development, we have successfully established an HDFS cluster deployed alongside our distributed DAPHNE Runtime. The cluster comprises of a central master node, referred to as the 'coordinator' – in a manner analogous to the NameNode in HDFS parlance – and multiple worker nodes, serving as DataNodes. The currently implemented methods are:

- *Connection Establishment Method*: A robust mechanism to initiate a stable connection with the HDFS, facilitating subsequent file operations.
- *File Read Method*: Enables the reading of files stored within the HDFS, ensuring seamless data retrieval operations.
- *File Write Method*: Provides the functionality to write data to files in HDFS, an essential operation for data persistence.
- *Block Write Method*: A specialized method to write data blocks to the HDFS with specific size parameters and a predefined replication factor, enhancing data redundancy and reliability.
- *CSV-to-DenseMatrix Method*: This method efficiently reads CSV files from HDFS and converts them into DAPHNE *DenseMatrix* format, optimizing for in-memory data manipulation.
- *DenseMatrix-to-CSV Method*: Complements the previous method by converting and writing DAPHNE *DenseMatrix* objects back to CSV format in HDFS, enabling both storage and interchange of processed data.
- *CSV Serialization Method*: A method dedicated to reading and serializing .CSV files from the HDFS into DAPHNE objects, streamlining the process of data transformation and preparation for analysis.
- *ByteStream Deserialization Method*: This method deserializes an HDFS bytestream, which currently is assumed to contain matrix data, into DAPHNE *DenseMatrix*, allowing for the reconstruction of matrix data into a usable format for our ecosystem.

Currently, what remains to be implemented is the precise mapping of DAPHNE matrix rows to the respective bytes within the HDFS bytestream. This mapping is critical for leveraging HDFS's inherent data locality features. For instance, the rows from 4 to 20 in the DAPHNE matrix correlate to bytes 36 to 122 in the HDFS bytestream. Realizing this mapping allows us to harness the full potential of HDFS's data distribution without necessitating alterations to the foundational HDFS code. By doing so, we can optimize data access patterns and performance, ensuring that our system can efficiently process large datasets by reading only the necessary data from HDFS, thereby reducing I/O and network overhead.

2.3 Runtime Communication Advances

2.3.1 Communication Backends

A significant enhancement in DAPHNE's communication capabilities is the incorporation of MPI (Message Passing Interface). MPI enables efficient communication and coordination with DAPHNE's distributed workers, making it an ideal choice for complex, high-performance computing tasks. We use the blocking implementation of MPI which means each message needs to be received by the recipient (worker) before the sender (coordinator) can send additional messages to the rest of the cluster. To overcome this constraint during data distribution, we utilize threads to concurrently transfer data to multiple workers.

However, it is worth noting that the advantages of MPI come hand in hand with certain complexities and challenges. Deploying MPI can be more intricate compared to traditional communication mechanisms, such as gRPC. Users may encounter hurdles during the installation and configuration of MPI libraries, which can vary depending on the specific computing environment and system configurations.

Moreover, MPI's communication model introduces a different set of programming complexities for DAPHNE developers and contributors. Unlike the more straightforward request-response pattern of gRPC, MPI operates on a lower level, requiring explicit message passing and synchronization. While this may initially pose challenges, it also provides a significant improvement in terms of fine-grained control over distributed processes. This level of control empowers DAPHNE developers to optimize performance with precision, making it particularly advantageous for those seeking a deeper understanding of parallel programming.

DAPHNE can be launched using MPI with the mpi command "`mpirun`" and providing the flag "`--dist_backend=MPI`" to the daphne binary (default distributed backend is MPI so the "`--dist_backend`" flag is optional in this case). An example with 4 instances of DAPHNE communicating with MPI is displayed below (one of them will be the coordinator and three will be distributed workers):

```
mpirun -n 4 ./bin/daphne –distributed --dist_backend=MPI ./script.daph
```

In addition to MPI, we have recently introduced support for the synchronous gRPC version, complementing our existing asynchronous gRPC capabilities. In the past, we have experienced performance issues with asynchronous gRPC. However, it is worth mentioning that asynchronous gRPC gives significantly more responsibility to the DAPHNE developer to manage resources. As such, there is a need for more experimentation/development in order to fully utilize this specific implementation.

Synchronous gRPC facilitates simpler, blocking communication patterns, which can be beneficial in scenarios where tight control over request-response interactions is necessary. In synchronous gRPC, calls to distributed workers are made in a blocking manner, meaning that a call will pause and wait for the response before proceeding. To ensure that this blocking nature does not hinder the overall system's performance, we have implemented a threading mechanism. By leveraging threads, DAPHNE can concurrently make calls to distributed workers, effectively avoiding any bottlenecks that might occur when waiting for responses.

The introduction of synchronous gRPC also has the advantage of simplifying the codebase. It offers a more intuitive and linear approach to handling requests and responses, making it accessible to new contributors and developers who may be less familiar with the intricacies of asynchronous programming. Moreover, gRPC is responsible for handling resources for requests and responses making it behave in a more reliable and performant manner.

Similarly to the existing asynchronous gRPC implementation, to start a distributed computation using sync-gRPC, we must first launch distributed workers executable that use synchronous-gRPC workers. Note that, since we now support more than one way to start a gRPC worker, we now pass an additional flag specifying which implementation of gRPC to use:

```
./bin/DistributedWorker --dist_backend=sync-gRPC IP:port
```

We must first set the environment variable DISTRIBUTED_WORKERS and then we invoke DAPHNE:

```
export DISTRIBUTED_WORKERS=IP:Port,IP2:Port2,…
./bin/daphne --distributed --dist_backend=sync-gRPC ./script.daph
```

2.3.2 Message Fragmentation

In conventional communication frameworks there exists a strict upper limit on the size of messages, typically around 2GB. However, the nature of DAPHNE often demands the transmission of significantly larger data volumes. To overcome this limitation, we have extended our communication operations to accommodate larger message sizes with *message fragmentation*. This mechanism enables us to break down oversized data payloads into smaller, manageable *segments* that can be effectively transmitted and subsequently reconstructed by distributed workers. This concept is seamlessly facilitated by our recently implemented serialization library, as mentioned earlier.

With our serialization library, we create fragments that are then transmitted to the intended recipients. Each fragment is accompanied by sufficient metadata and header information to facilitate proper reassembly. We provide an additional flag to DAPHNE that specifies the maximum fragment size that can be used, in case there are limited memory resources:

`--max-distr-chunk-size=N` (where N in bytes).

2.4 Execution Engine

2.4.1 Metadata Support

In [D+22a] we described DAPHNE's data object metadata. Local and distributed data structures store metadata, including shape, sparsity, symmetry, and data placement. This information can be known at compile-time or determined at runtime, providing flexibility. Metadata helps optimize operations, such as choosing efficient algorithms for sorted data. Data placement is crucial, especially in hybrid runtime plans utilizing distributed workers and heterogeneous hardware. Matrices and frames reference data across various devices, and for federated data, metadata includes partition details and data location at the coordinator.

We extended DAPHNE's metadata implementation for the distributed runtime by incorporating support for both gRPC and MPI communication frameworks. Each framework necessitates distinct metadata due to their unique requirements. For instance, gRPC relies on worker addresses (IPs and ports), while MPI uses rank ids. These metadata implementations also include details about the distribution of rows and columns among workers, along with other runtime information.

2.4.2 Kernel updates

2.4.2.1 Eigen

Eigenvalues and eigenvectors are foundational concepts in linear algebra with pivotal applications in various computational algorithms. Eigenvalues and eigenvectors are crucial in numerous algorithms that require dimensionality reduction, optimization, and solving systems of differential equations. For instance, the Principal Component Analysis (PCA) algorithm, which is widely used for reducing the dimensionality of large data sets by transforming them into a new set of variables, relies on eigenvalues and eigenvectors to identify the principal components. Other algorithms that utilize these concepts include Singular Value Decomposition (SVD), Latent Semantic Analysis (LSA), Linear Discriminant Analysis (LDA), many Spectral Clustering algorithms, etc.

To facilitate these operations within our system, we have integrated the *Eigen C++ library* with DAPHNE. This allows us to construct a specialized kernel capable of computing the eigenvalues and eigenvectors of a given Daphne matrix. Leveraging this kernel, we have developed a custom operator (a DaphneOp), which enables Daphne users to compute eigenvalues and eigenvectors directly within the DaphneDSL. This integration empowers users to incorporate these calculations seamlessly into their algorithms.

2.4.2.2 I/O and data-format Support

In order to support a DaphneDSL PageRank implementation, we needed to extend the implementation and specialization of various runtime kernels (*MatMul, Gemv*), especially with regard to supporting CSR matrices for various operations. We have also added kernels for:

- Outer binary operations (generalized outer product) on DenseMatrix.
- Cumulative aggregation on DenseMatrix.
- CSV file writer for frames.

2.4.2.3 Cuda-specific Kernels

During this past year, the support of CUDA instructions in the DAPHNE runtime has seen several advances in functionality as well as bug fixes and usability improvements.

The Connected Components algorithm, which is featured in several deliverables and publications as a simplified use case scenario, has now relevant operations implemented with a CUDA kernel. Some of the needed operations were already there, whereas the newly added ones contain a number of element-wise operations, row aggregates and full aggregates.

Additional CUDA operations that were implemented are the fill instruction and a launcher instruction for run-time generated and compiled fused operations. The former is used to fill data objects with single value data directly in GPU memory, thereby avoiding the need to

generate that data in host memory and then transferring it to the devices' memory in a separate step. The latter is used to launch CUDA kernels that have been compiled at run-time if the DAPHNE compiler has been able to fuse operations according to certain patterns. This "codegen" operation is discussed further in Deliverable D7.3.

On the improvements and bug fixes side, the DNN operations were fixed to accommodate refactorings in DAPHNE's code base and the transpose operation now correctly handles vector transpose in case of vectorized (tiled) processing.

Lastly, the use of Docker-based container images with CUDA support avoids the need to have all the correct versions of third-party dependencies installed on the host system, which improves ease of use considerably.

2.4.3 Monitoring/tracing support

We added basic monitoring support for the Daphne Runtime System via the PAPI library[7]. PAPI lets the DAPHNE runtime interact with the PMU / PMC (performance monitoring unit / counters) of the underlying CPU, in order to extract fine-grained architectural and micro-architectural event information, e.g., cycles, instructions, cache misses, branch misses etc. It also provides support for other monitoring plugins (e.g., GPU monitoring, power consumption, etc.).

We implement this support via two monitoring kernels (`StartProfiling` and `StopProfiling`), which can be used to instrument either other kernels or the entirety of the DAPHNE script. We currently use these kernels for the latter, injecting them at the start and end of the execution of the user-provided DAPHNE script. These kernels are enabled by a user-provided CLI option (`--enable-profiling`) and then configured by setting the relevant PAPI environmental variables (e.g., PAPI_EVENTS='cycles, instructions').

Additionally, we support exposing these collected metrics, as well as coarse-grained timing information (enabled by the `--timing` CLI option) to Prometheus, via the text file collector of the Prometheus node exporter. This is work done for our demonstrator UI [V+23] and is currently not included in the artifact.

2.4.4 Compiler support

In [V+22], we described a limitation where pipelines with duplicate inputs within the pipeline could not be executed. This limitation has been addressed by enhancing the optimization pass responsible for generating distributed pipelines. If a pipeline input appears multiple times, we simply remove the duplicates and create references to the same argument within the intermediate representation (IR).

However, we still do not support pipelines in cases where the arguments exist multiple times but are distributed in different schemas across workers (e.g., distributed by rows and columns).

[7] https://icl.utk.edu/papi/

3 Daphne Runtime v2 Prototype

3.1 Artifact Access and use

The DAPHNE DSL runtime prototype described in this deliverable is publicly accessible as a snapshot of the DAPHNE development repository (created in November 9th, 2023) under the following link: https://daphne-eu.know-center.at/index.php/s/arrNCQGiSmKid5s

Note that the DAPHNE development repository is publicly available at https://github.com/daphne-eu/daphne under Apache License v2.0.

In [V+22] we provided detailed guidelines on how to build DAPHNE and run a simple DSL script. The same steps can be repeated to build the latest version of DAPHNE provided in the current artifact. MPI can now be used with DAPHNE, however is not installed by default. After installing the required dependencies on the system, we need to re-build DAPHNE and provide the MPI flag:

```
$ ./build.sh --mpi
```

One additional important change is that now we support more than one communication framework, therefore we need to specify the distributed backend when running DAPHNE. This applies to the distributed gRPC workers as well, since we support two different versions of gRPC backends (synchronous and asynchronous).

To execute DAPHNE using different communication frameworks we use the "dist_backend" flag:

```
$ export DISTRIBUTED_WORKERS=IP1PORT1,IP2:PORT2

$ ./bin/daphne --distributed --dist_backend=sync-
gRPC ./scripts/examples/hello-world.daph
```

and in a similar way to start distributed gRPC workers:

```
$ ./DistributedWorker --dist_backend=sync-gRPC IP:Port
```

For MPI, we use the *mpirun* command to start multiple instances of DAPHNE:

```
$ mpirun -n 4 ./bin/daphne --distributed --
dist_backend=MPI ./scripts/examples/hello-world.daph
```

In Section 3.2 we present results from two DSL scripts: PageRank and Connected Components. Both of these algorithms can be found in the artifact at scripts/algorithms/. They require a graph as an input. The input used in the examples below is the Amazon product co-purchasing network[8] and can be found inside the artifact under "datasets" directory. Example use (for Connected Components):

[8] https://snap.stanford.edu/data/amazon0601.html

```
$. ./bin/daphne --select-matrix-
repr ./scripts/algorithms/components.daph G=\"datasets/amazon.mtx\"
C=\"outDistr.csv\"
```

For PageRank:

```
$ ./bin/daphne --select-matrix-
repr ./scripts/algorithms/pagerank.daph G=\"datasets/amazon.mtx\"
alpha=0.85 center=true scale=false maxiter=50
```

3.2 Evaluation Results

In this Section we present benchmarking results for the DAPHNE Runtime as included in the latest artifact.

3.2.1 Infrastructure and Inputs

We use the VEGA supercomputer for benchmarking. VEGA consists of a total of 960 nodes (including both CPU and GPU partitions) and uses Ceph. The CPU partition which we use has 768 nodes, each with a total of 256GB RAM and AMD Epyc 7H12 two socket processors. SLURM is used to submit jobs and allocate resources. For our experiments, we assign 80GB of RAM to each node. We are able to request multiple nodes as well as multiple CPUs per node, in various configurations. We evaluate the DAPHNE Runtime executing two different algorithms, Connected Components and PageRank.

In Table 1 and Table 2 we detail the different graphs that were used as inputs for each algorithm. All the datasets are publicly available. The Amazon sets refer to Amazon product co-purchasing network, June 01, 2003, SNAP Library, Stanford University. The other datasets (uk-2002, delaunay_n24, kmer_V2a and kmer_P1a) are downloaded from the SuiteSparse Matrix Collection[9].

Table 1: Inputs for Connected Components

Name	Dimensions (#nodes)	Non zero values	Disk size
Amazon x30	12101820	101621640	2GB
Amazon x100	40339400	338738800	6.8GB
uk-2002	18520486	298113762	4.7GB
delaunay_n24	16777216	50331601	0.8GB

Table 2: Inputs for PageRank

Name	Dimensions (#nodes)	Non zero values	Disk Size
kmer_V2a	55042369	58608800	0.98GB
kmer_P1a	139353211	148914992	2.6GB

[9] https://sparse.tamu.edu/

3.2.2 Experimental Evaluation

In this Section, we present a selected subset of benchmarking results in order to register the performance of the current runtime version under the dimensions of:

- Scalability of the local and distributed runtime over the number of available CPUs and workers respectively.
- Choice of different communication backends (gRPC and MPI).
- Message fragmentation (as described in Section 2.3.2) performance.
- DAPHNE serialization versus traditional CSV I/O.

All the reported datapoints are the average of three runs. Moreover, we note that the runtime uses the default STATIC scheduling for all these experiments.

3.2.2.1 Local Runtime

We first focus on the local runtime, comparing normal execution vs. the vectorized engine which leverages multiple cores in a system. Figure 5 shows results for the Connected Components algorithm while Figure 6 shows results for PageRank.

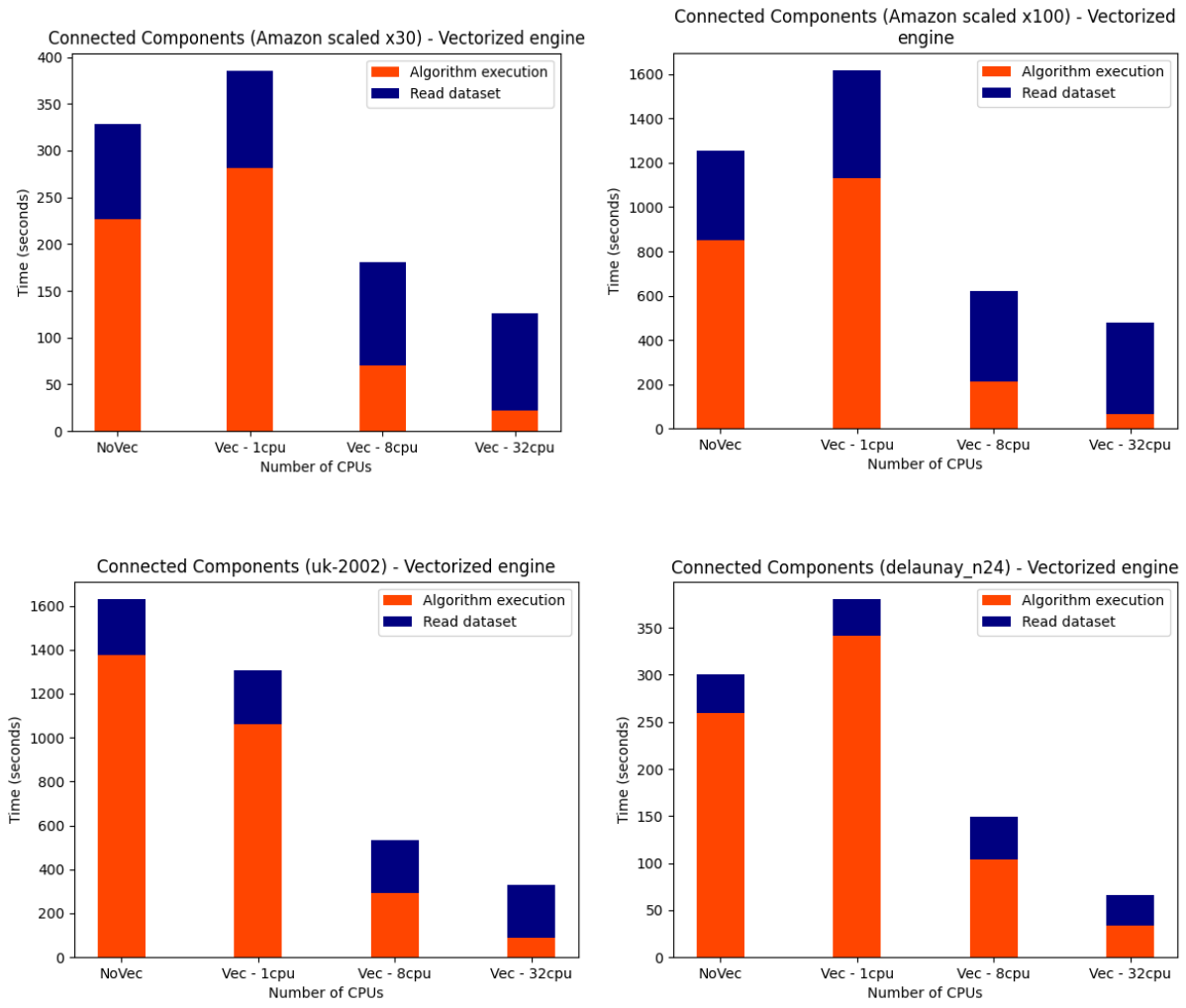


Figure 5: Local Runtime performance for Connected Components

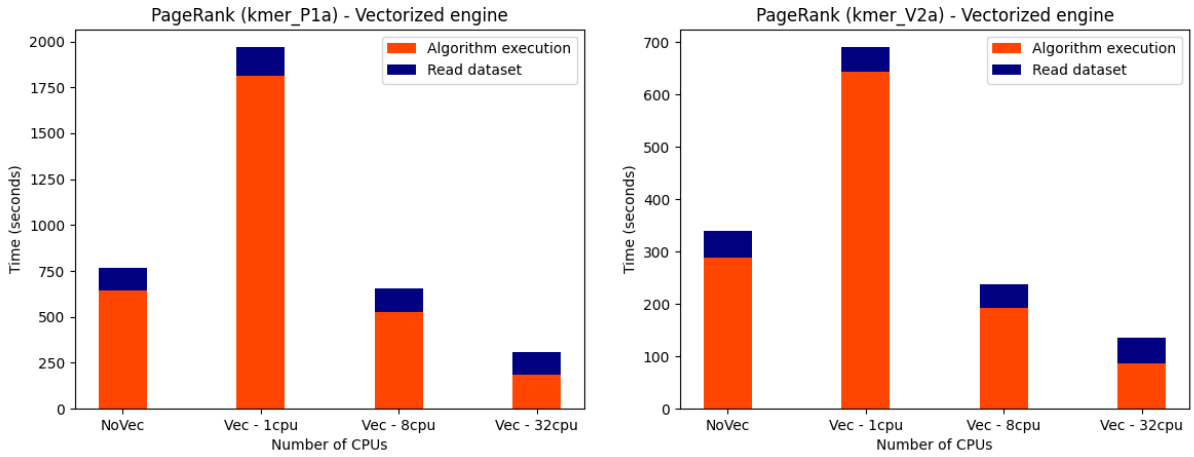
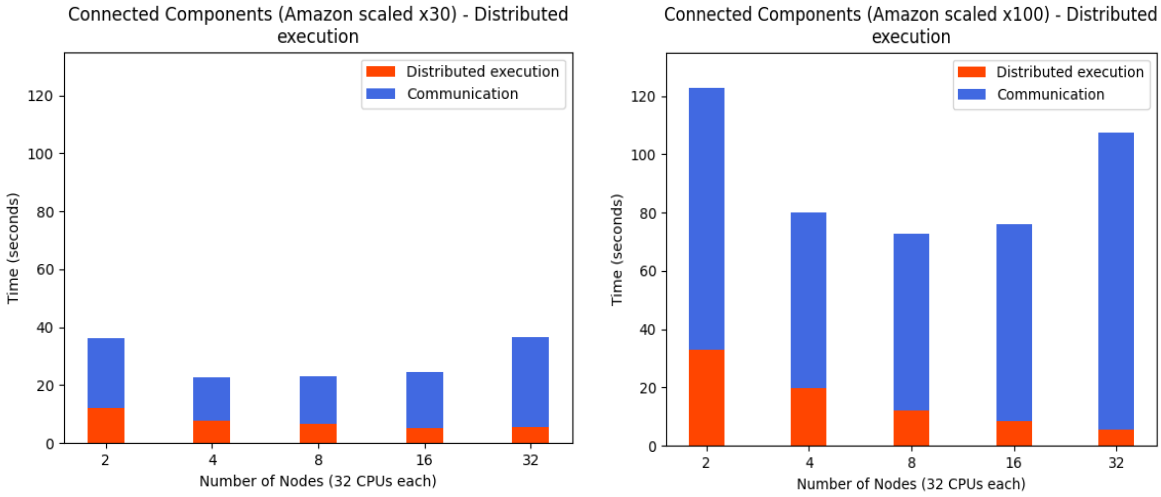


Figure 6: Local Runtime performance for PageRank

In both cases (algorithms), we notice how increasing the number of CPUs from 1 to 32 allows the vectorized engine to scale its performance, achieving a 17x speedup in the large Amazon graph and a 12x speedup in uk-2002 relative to the execution time for the Connected Components. For PageRank, the respective speedup reaches 13x (for kmer_P1a).

3.2.2.2 Distributed Runtime

In our next experiment, we present the distributed runtime with the gRPC backend, where each distributed node utilizes 32 CPUs. In the following plots, we describe time spent for communication and computation. Figure 7 shows results for the Connected Components algorithm while Figure 8 for PageRank.



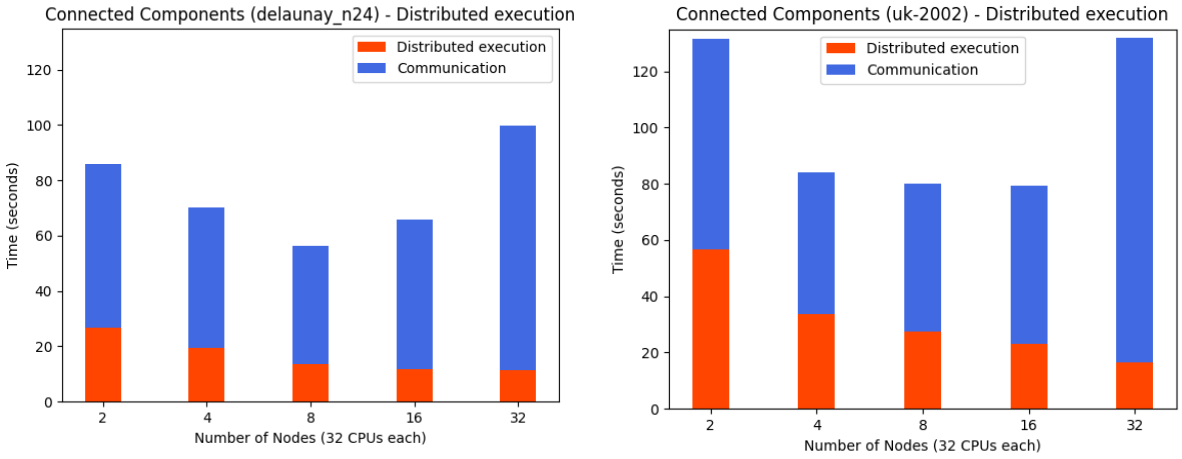


Figure 7: Distributed Runtime performance for Connected Components

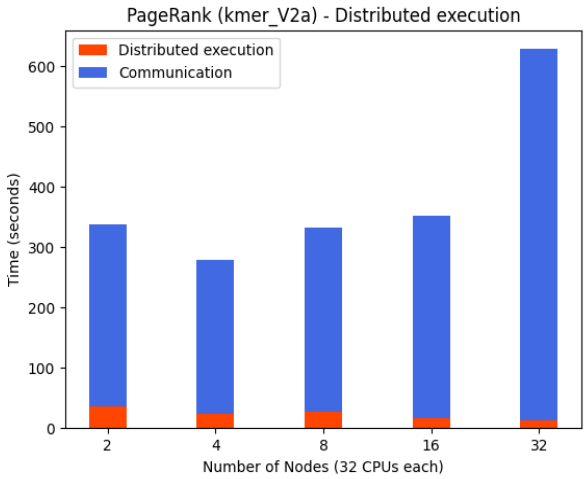


Figure 8: Distributed Runtime performance for PageRank

In all cases, the distributed mode successfully divides computation across available nodes, reducing the pure execution time between 3x and 12x (for the Connected Components) and between 7x and 11.5x (for PageRank) compared to the local runtime execution time. It is also evident that a "sweet-spot" in the optimal number of deployed DAPHNE worker nodes exists: Scaling up to that point (around 8 workers in these experiments) reduces the total execution time. Beyond that (and relative to the executed script), the communication overhead increases. These results also showcase that there exists plenty of room for improvement in reducing communication overhead via algorithmic and/or systemic methods in DAPHNE.

3.2.2.3 Communication backends

In the next experiment, we showcase the performance of gRPC and MPI used in the distributed runtime. We create a benchmark where we repeatedly call the *Distribute.h*[10] kernel for each communication backend in order to compare their performance. The dataset being distributed

[10]Artifact path: src/runtime/distributed/coordinator/kernels/Distribute.h

is delaunay_n24 (1.7G in memory, in CSRMatrix format), for 100 iterations. In addition, we use a fragment size of 10MB per message sent, from the coordinator to the workers. In Figure 9, we show the communication time for clusters of 2 up to 32 worker nodes.

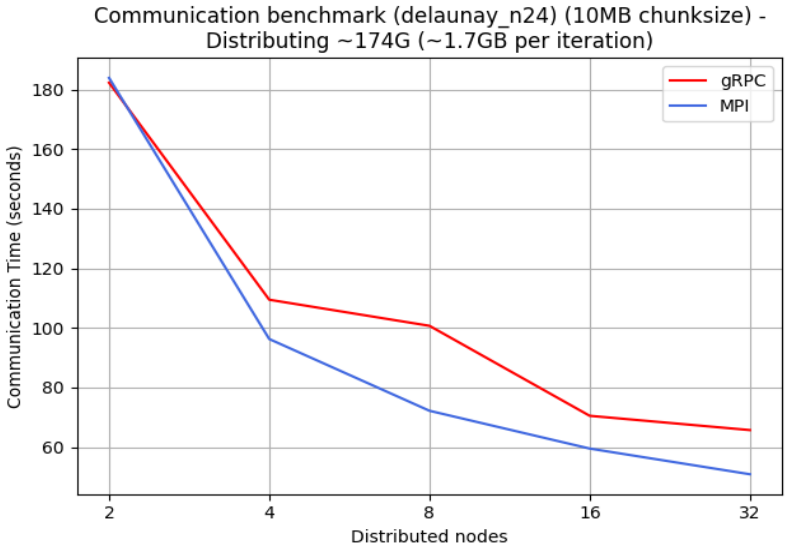


Figure 9: MPI and gRPC performance comparison

We notice how communication time is scaled as we increase the number of workers. As the cluster size increases, MPI proves to be more advantageous in terms of communication. Increasing the number of nodes results in less communication time, since, for both implementations, we spawn a number of threads, equal to the number of nodes, each one responsible for transmitting data to a worker node. Moreover, the size of data received by each node is decreasing as we distribute data to more nodes.

3.2.2.4 Message Fragmentation

In this benchmarking scenario, we study how the performance of DAPHNE distributed runtime, and specifically communication time, is affected by different fragment sizes. We range the user-defined size from 50KB up to 1GB, and execute the Connected Components algorithm over an 8-worker (8 CPUs each) cluster. Results are shown in Figure 10 for both the gRPC and MPI backends.

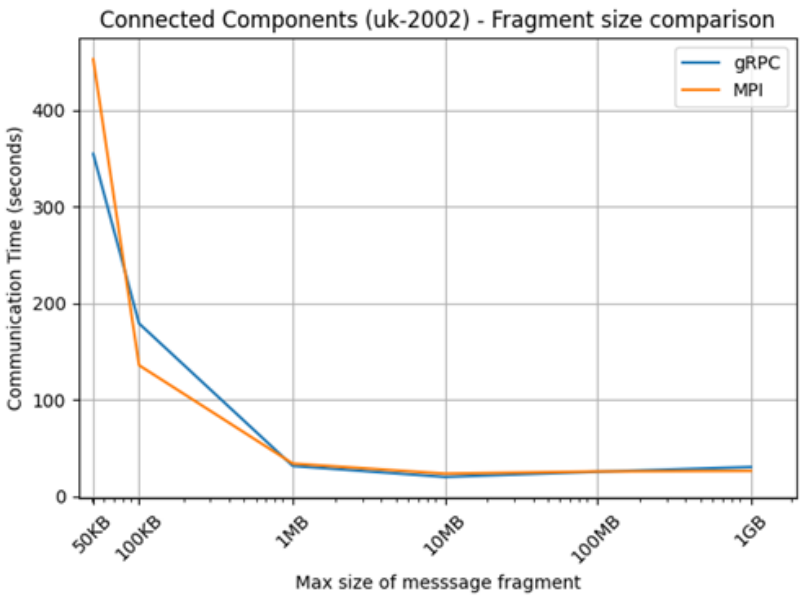


Figure 10: Message fragment size effect on communication time

Increasing the fragment size from very small values is highly beneficial to the communication time, resulting in more than ten-fold reduction. Yet, after some value (around 1MB in our experiments) the cost only decreases slightly, with the communication stabilizing or even marginally increasing – its minimum value taken around a 10MB fragment size for both backends. These findings will be further investigated in the future: The current scale of our algorithms/inputs require small processing times which could easily produce misleading conclusions.

3.2.2.5 DAPHNE Serialization

In the final experiment, we compare the I/O performance for DAPHNE Objects, using our newly implemented Serialization library vs. the traditional CSV representation. We create a random generated matrix (10K rows by 50K columns), with a size of 4.2GB stored as CSV and 3.8G stored as DAPHNE binary format. Figure 11 shows the time it takes for a read or write operation of this matrix using our serialization method compared to the CSV ones.

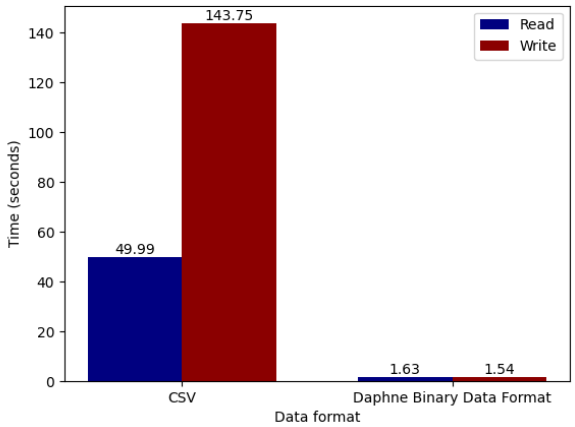


Figure 11: DAPHNE object vs. CSV Serialization performance

DAPHNE serializer proves faster by a factor of about 30x in reading and 93x in writing. The performance difference is due to that fact that the CSV implementation has to iterate for all matrix elements in order to read or write them, whereas DAPHNE serializer simply writes or reads bytes from disk, in the same format as stored in memory without the need for parsing or conversions to text format. Nevertheless, similarly to the previous paragraph, this finding will be further investigated and tested against our constantly updated serialization functions. Many factors, such as the deployment platform's filesystem (Ceph in the VEGA case) play a big role in such benchmarking results.

4 Conclusions & Future Work

In this deliverable, we described the current design and implementation status of the DAPHNE Runtime. During the last year, there has been significant progress that has been registered in multiple dimensions of the system: a) In commencing the integration with a new storage backend, HDFS; b) in optimizing I/O in DAPHNE Runtime, by providing an efficient serialization library; c) in providing support for existing kernels as well as introducing new ones, both for more ML operations as well as ones that support an increasing number of formats; d) in adding MPI as a possible communication backend, and optimizing both existing options via message fragmentation, and e) in enhancing the runtime's usability by providing monitoring and tracing support.

In the upcoming months, we plan on performing and testing the full integration with HDFS, as well as benchmark its performance gains/conditions under which this is possible. Integrating with HPC-preferred filesystems (such as Lustre [Lu]) is a desirable direction. We also plan on investigating the use of the NVIDIA Collective Communication Library (NCCL) as a possible communication backend, but, most importantly, considering a *hybrid* communication technique in DAPHNE Runtime, namely one that utilizes existing backends adaptively.

References

[B21] M. Boehm. DAPHNE Deliverable D2.1 "Initial System Architecture", 2021.

[D+22] P. Damme et al. "An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines". In CIDR 2022.

[D+22a] P. Damme et al. DAPHNE Deliverable D2.2 "Refined System Architecture", 2022.

[Ha] The Apache® Hadoop® project. https://hadoop.apache.org/

[K21] V. Karakostas et al., DAPHNE Deliverable 4.1 "DSL Runtime Design", 2021.

[L+20a] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law." 2020.

[L+20b] S. Li et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training." 2020. In Proceedings of the VLDB Endowment, 2020. Vol. 13, No. 12, pp. 3005-3018.

[L+21] C. Lattner et al., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 2-14.

[Lu] Lustre® file system. Lustre

[V+22] A. Vontzalidis et al., DAPHNE Deliverable D4.2 "DSL Runtime Prototype", 2022.

[V+23] A. Vontzalidis et al., "DAPHNE Runtime: Harnessing Parallelism for Integrated Data Analysis Pipelines". In the 29th International European Conference on Parallel and Distributed Computing (Euro-Par), 28 August – 1 September 2023, LIMASSOL, CYPRUS (2023)