D4.4 Final DSL Runtime Prototype

DAPHNE

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

Version 1.1

PUBLIC

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

Document Description

DAPHNE is an open and extensible system infrastructure for Integrated Data Analysis (IDA) pipelines, including language abstractions, compilation and runtime techniques, multi-level scheduling, hardware accelerators, and computational storage. Previous deliverables described the initial [D2.1] and refined [D2.2] DAPHNE system architecture, while the initial design of the local and distributed DAPHNE runtime system was described in D4.1 and its updates in D4.3.

In this deliverable we describe the updates in the runtime system during the last year. We also demonstrate the use of the DAPHNE runtime by sharing a snapshot of the DAPHNE prototype, and provide instructions how to execute its new features.

This prototype and document are the result of the collaborative work that is performed by all consortium partners that participate in WP4 "DSL Runtime and Integration".

D4.4 Final DSL Runtime Prototype			
WP4 – DSL Runtime and Integration			
Type of document	D	Version	1.1
Dissemination level	PU	Project month	48
Lead partner	ICCS		
Author(s)	Aristotelis Vontzalidis (ICCS), Dimitrios Tsoumakos (ICCS), Stratos Psomadakis (ICCS), Konstantinos Bitsakos (ICCS), Vassiliki Kostoula (ICCS), Jonas H. Müller Korndörfer (UNIBAS), Quentin Guilloteau (UNIBAS), Florina M. Ciorba (UNIBAS), Patrick Damme (TUB)		
Reviewer(s)	Patrick Damme (TUB), Vytautas Jancauskas (DLR)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Initial outline	Dimitrios Tsoumakos
V0.2	Updated content	Dimitrios Tsoumakos, Aristotelis Vontzalidis, Stratos Psomadakis, Konstantinos Bitsakos, Vassiliki Kostoula
V0.3	Updated Sections 2.3 and 3.2	Dimitrios Tsoumakos, Aristotelis Vontzalidis, Stratos Psomadakis, Vassiliki Kostoula, Jonas H. Müller Korndörfer, Quentin Guilloteau, Florina M. Ciorba
V0.4	Updated Section 2.1.2	Dimitrios Tsoumakos
V0.5	Updates with spelling & various errors after reviews	Dimitrios Tsoumakos
V0.6-V0.8	Updates in all sections based on review comments	Dimitrios Tsoumakos, Aristotelis Vontzalidis, Stratos Psomadakis, Vassiliki Kostoula
V0.9	Updates in formatting and missing references	Dimitrios Tsoumakos
V1.0	Updates in Section 2.2.1 (Kernels)	Patrick Damme
V1.1	Polishing document	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	Graphics 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

1.1 Overview

DAPHNE (DAta Processing and High-performance computing for integrated data analysis pipelinEs) is a system designed to meet the rising demand for integrated data analysis (IDA) pipelines that merge data management, high-performance computing (HPC), and machine learning (ML). DAPHNE aims to overcome integration challenges between various systems and hardware constraints to facilitate seamless execution of complex IDA pipelines. This system is built on MLIR (Multi-Level Intermediate Representation), which allows integration with existing applications and runtime libraries and supports extensibility for specialized data types, hardware-specific compilation, and custom scheduling algorithms. Key components of the DAPHNE system are:

- **DaphneDSL** and **DaphneLib**: Users can define their workflows in either DaphneDSL, a domain-specific language resembling Python and R, or DaphneLib, a high-level Python API that compiles scripts into executable plans.
- **Compiler**: DAPHNE's compiler pipeline generates execution plans by breaking down high-level operations into specialized operations for different devices.
- **Execution Engine (Runtime)**: The runtime executes the plans created by the compiler in either local or distributed environments, leveraging parallelization for performance optimization. At the core of the runtime, the **Vectorized Execution Engine** enables fine-grained parallelism by allowing operator fusion and vectorized execution (as designated by its compiler component counterpart) across heterogeneous hardware such as CPUs, GPUs, and distributed nodes.

The DAPHNE runtime is structured to support efficient execution of user-defined workflows and integrates the following components:

- **DAPHNE Kernels**: These are device-specific C++ kernels executing core operations, optimized for distributed or accelerated hardware. The runtime uses specialized kernels for diverse operations and provides flexibility for handling dense and sparse matrices.
- **Data Structure Support**: The runtime handles various data formats, including dense and sparse matrices and frames with column-oriented storage, both locally and in distributed environments.
- **Communication Framework**: The runtime integrates with communication frameworks such as gRPC and MPI to handle data and code transfer across nodes. This modular design allows flexibility in selecting the optimal communication backend based on the use case.
- **I/O Management**: The runtime is optimized for efficient I/O operations, supporting multiple formats (CSV, Apache Arrow, Matrix Market) and a custom DAPHNE-specific file format for faster data serialization and transfer.

1.2 Summary of Updates

The DAPHNE Runtime system has been significantly enhanced with integrations and optimizations for high-performance I/O management. File system integrations include the finalization with the Hadoop Distributed File System (HDFS[1]) and a new integration with Lustre[2], each offering distinct advantages for big data and distributed machine learning workflows. The HDFS integration allows efficient, scalable data handling with high throughput and fault tolerance, leveraging data locality to minimize latency and facilitate parallel processing. A private cluster setup verified HDFS's improvements in both read and write performance for distributed datasets. Lustre integration, aimed at HPC contexts, brings parallel processing with data striping across Object Storage Targets (OSTs), reducing latency for large-scale applications. Using a mounted Lustre client in a DAPHNE container enables seamless access to distributed data.

Enhancements also include NUMA-aware data placement, optimizing memory locality in NUMA architectures to reduce access latency. This improvement particularly benefits dense matrix computations, showing substantial performance gains in both native and virtualized environments. New filetype support now includes Coordinate List (COO) for sparse matrices, improving memory efficiency for ultra-sparse data. Additionally, Technical Data Management Streaming (TDMS) format support was added for high-performance data acquisition in technical and engineering applications, with efficient readers and writers mapping TDMS structures to DAPHNE's data formats. Finally, a number of further improvements that include new kernels, kernel time measurement, systematic error message generation, and efficient data exchange between DaphneLib and Python libraries have been delivered. Together, these enhancements make DAPHNE more robust for complex data processing across diverse, large-scale environments.

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 Final version 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.

[1] https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[2] https://www.lustre.org/

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 D4.3 [D4.3] in November 2023).

2.1 Runtime I/O

2.1.1 Integration with HDFS

2.1.1.1 Why integrate with HDFS

In [D4.3] we described our effort (until the time of the deliverable) to integrate the Hadoop Distributed File System (HDFS) filesystem with the Daphne Runtime. HDFS integration is required for several reasons, especially in the context of large-scale data processing and machine learning.

Firstly, the specific system is *highly scalable*. HDFS is built to work with large amounts of data and is scalable as the amount of data increases. This makes it an essential part of growing datasets in big data and machine learning execution and management environments.

Another important factor is *ecosystem support*. HDFS is compatible with multiple important big data tools and frameworks like Hadoop, Spark, and Hive. This compatibility means that machine learning applications can run in a stable and adaptable environment that can leverage the many tools available of the ecosystem.

A very important feature of HDFS is *data locality*. Data locality is important for reducing I/O costs because it minimizes the need to move large datasets across a network. In HDFS, data is stored on the same nodes that perform the computations. This allows tasks to access data locally rather than over the network, significantly reducing I/O wait times and network congestion, which in turn boosts overall processing speed and efficiency.

As for reliability, HDFS provides fault tolerance via *data replication*. HDFS replicates data by storing multiple copies of each data block across different nodes in the cluster. A file served from HDFS is split into blocks, and each block is stored on multiple nodes according to the configured replication factor (default is three). This redundancy ensures data reliability and availability, allowing the system to recover from node failures by accessing replicated blocks on other nodes.

Last but not the least, HDFS offers *high throughput for large data sets*. It is designed for high throughput access and is ideal for use in big data and machine learning applications. This guarantees that large data sets can be read and written effectively, which is a characteristic of large data operations common in these disciplines.

2.1.1.2 HDFS Architecture – C++ API

HDFS has a coordinator-worker architecture with one coordinator node called *NameNode* and multiple worker nodes called *DataNodes*. Such division of roles helps in the management and distribution of data within the system in an efficient manner. As for the data storage, the files in HDFS are divided into data blocks with a default size of 128 MB and these blocks are located in DataNodes. This distribution enables efficient management of the load and the amount of parallel processing required.

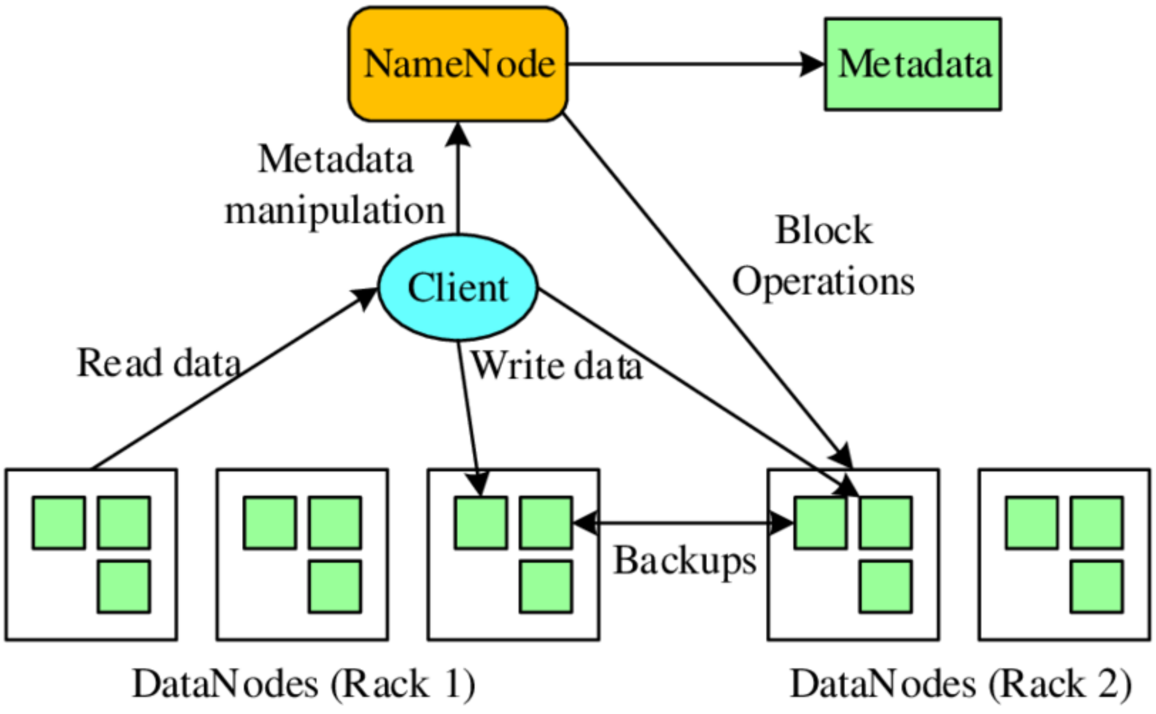

Figure 1: HDFS Architecture

To achieve fault tolerance, each data block is stored in multiple DataNodes in the Hadoop cluster. This replication strategy offers protection, as it ensures that the system does not lose data in case of hardware failure. The NameNode is responsible for managing metadata and the namespace of the file system. It tracks the data blocks and their locations so that access to files remains fast and easy for users and applications. These are described in Figure 1[3].

2.1.1.3 Libhdfs3

The `libhdfs3` library[4] offers a C/C++ client that enables native and efficient access to HDFS. Optimized for performance, it also manages resources when it is embedded in C and C++ applications. Another advantage of `libhdfs3` is that it does not require Java dependencies

[3] https://www.researchgate.net/figure/HDFS-Architecture-Diagram_fig2_376718504
[4] https://github.com/erikmuttersbach/libhdfs3

and thus avoids the issues related to jars and JVM libraries. This makes it easier to integrate into C/C++ projects than other options that involve Java-based elements.

Also, `libhdfs3` provides asynchronous I/O operations, which means that it is possible to read data without blocking. This feature improves the application's interactivity and the number of requests per unit of time, which makes it ideal for applications that require massive data processing.

2.1.1.4 Integration

The integration between DAPHNE and HDFS is designed to support seamless handling of large datasets and distributed computing tasks. In terms of the workflow specification, the integration facilitates *data upload* in both the *DAPHNE binary format* and the widely *used CSV file format*. This flexibility ensures compatibility with different data sources. Moreover, it provides distributed read and write capabilities, allowing for efficient management of large datasets across multiple nodes.

DAPHNE is employed for distributed data processing and machine learning tasks, where it is optimized for high performance on big data workloads through its architecture and, specifically, distributed and vectorized runtime. After data processing is complete, the *data save* functionality enables users to store the processed results in either DAPHNE binary format or CSV format, depending on their requirements. It also supports distributed write operations, offering users the ability to specify output destinations, be it HDFS or local storage.

Several implemented methods underpin this integration. For example, the system supports reading and writing in DAPHNE's native binary format with optimized efficiency. Similarly, it handles CSV file formats for input and output operations, maintaining compatibility with standard data formats. The integration also includes distributed read/write operations for both DAPHNE binary and CSV formats, ensuring scalable performance for large datasets in distributed environments. This combination of features makes the DAPHNE-HDFS integration highly effective for big data processing tasks.

Libhdfs3 provides straightforward file handlers for manually reading and writing data. Instead of relying solely on predefined functions like complete CSV readers or writers, these file handlers enable us to directly interact with the data. This approach offers a high degree of flexibility, allowing us to fine-tune operations. By leveraging this level of control, we can optimize data processing workflows for particular use cases, such as selectively reading subsets of data or implementing custom serialization techniques. This granularity complements the distributed and high-performance design of DAPHNE, ensuring that the integration remains both robust and adaptable to a wide range of big data scenarios.

2.1.2 Integration with Lustre

2.1.2.1 Lustre Overview

Lustre[5] is a high-performance, POSIX-compliant, open-source parallel distributed file system designed primarily for large-scale computing environments. It is widely used for applications requiring extensive data processing, such as scientific research, AI, and big data analytics. Lustre is renowned for its scalability, with the ability to handle petabytes of data and thousands of clients, making it suitable for high-performance computing (HPC) clusters and supercomputing environments.

The Lustre filesystem is built on a client-server architecture, split into specialized components that manage metadata and storage for high efficiency and scalability. Key components are:

- **Metadata Server (MDS)**: Manages metadata operations (e.g., directory structure, permissions).
- **Object Storage Server (OSS)**: Handles the actual file data, storing it across one or more Object Storage Targets (OSTs).
- **Management Server (MGS)**: Central repository for configuration data, which is shared among all Lustre clients.
- **Clients**: Endpoints that mount the Lustre filesystem, appearing as a POSIX-compliant mount point on the client OS.

These are pictorially described in Figure 2[6]:

Lustre achieves performance through *data striping*, where files are divided across multiple Object Storage Targets (OSTs) in chunks called stripes based on parameters like `stripe_count` and `stripe_size`. The stripe count determines how many OSTs will be used to store the file data, while the stripe size determines how much data will be written to an OST before moving to the next OST in the layout. Each object contains chunks of data from the file, and chunks are written to the file in a circular round-robin manner. When the chunk of data being written to a particular object exceeds the stripe size, the next chunk of data in the file is stored on the next object. The client selects the striping layout and determines the `stripe_count` and `stripe_size` parameters which are fixed once the file is created.

[5] https://www.lustre.org/

[6] https://wiki.lustre.org/File:Lustre_components.png

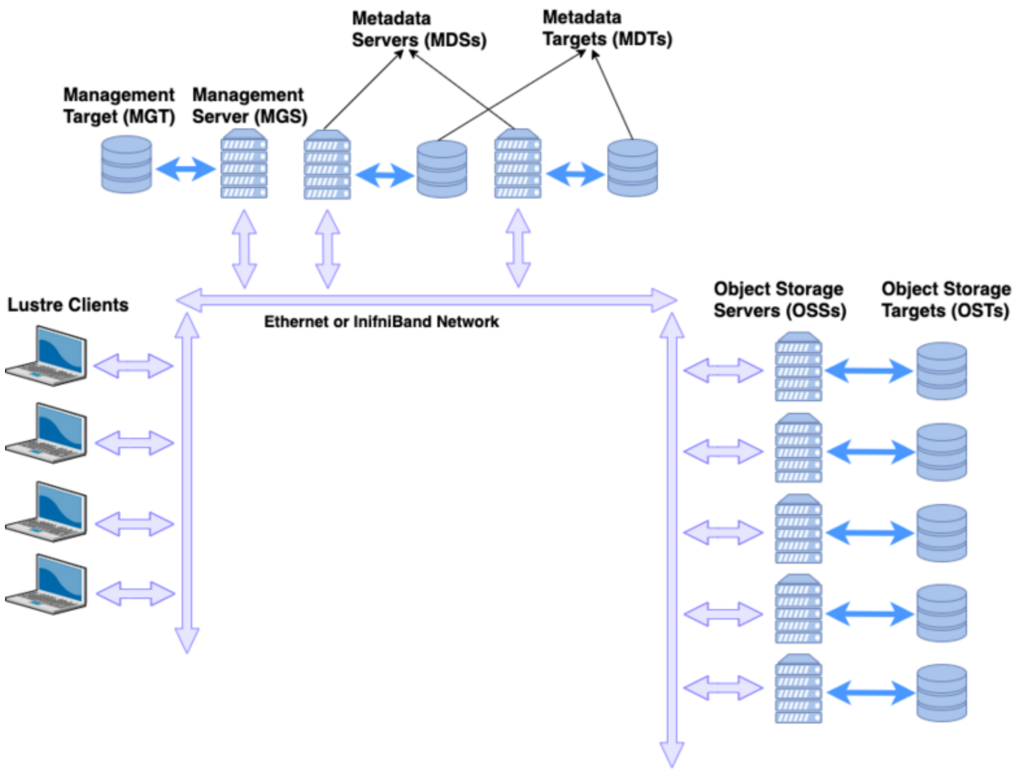

Figure 2: Lustre FS System Architecture

As an example, consider the file layouts shown in Figure 3[7] for a simple file system with 3 OSTs residing on 3 different OSS nodes. Note that OSC and LOV correspond to Logical Object Volume and Object Storage Client (OSC) components. Requests for data are routed to LOV which acts as an abstraction layer for all of the OSC components. There is an OSC component for each OST target in the file system.

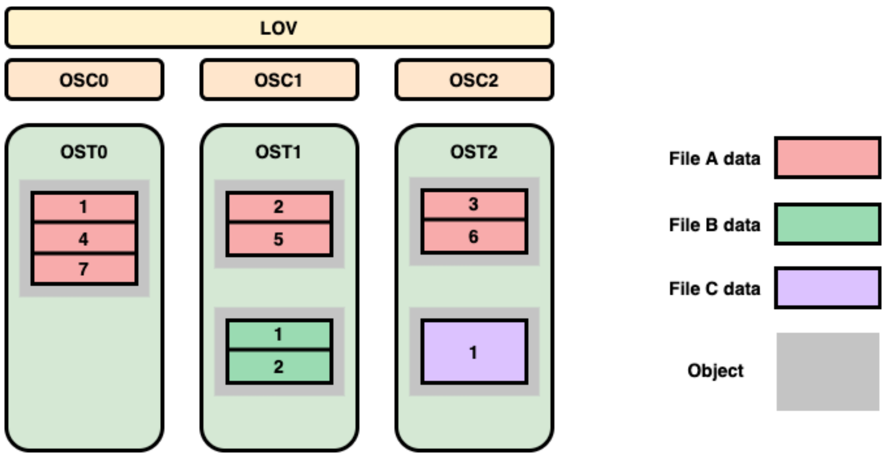

Figure 3: Normal file striping in Lustre

[7] https://wiki.lustre.org/Understanding_Lustre_Internals

2.1.2.2 Integration with DAPHNE

For DAPHNE to support I/O to Lustre FS, a Lustre client that mounts the filesystem is required. Then, a DAPHNE docker container is deployed on top of that client. The container mounts the external Lustre mount point inside the container, allowing DAPHNE kernels to interact with the Lustre filesystem. This separates the Lustre Client modules from the DAPHNE containers, meaning that we can easily use DAPHNE on existing Lustre infrastructures. It also minimizes the dependencies that are installed on the DAPHNE container, since it only needs the C API library to interact with the mounted filesystem. Our integration work utilizes the `liblustreapi` C API. This is the official C API which is part of the Lustre source code.

Currently, Read and Write kernels have been implemented for CSV files and serialized DAPHNE objects. The kernels are implemented for both the local and the distributed runtime. The Lustre C API library is used to create a file and set the striping layout. Since Lustre is POSIX-compliant, regular system calls can be used to obtain file descriptors for the Lustre files after file creation. In turn, these descriptors are used to read and write data with system calls such as `pread` and `pwrite` respectively.

Our implementation of the Read and Write kernels is similar to the ones for the local filesystem. One important compromise that was made in our initial implementation for CSV files is that each value needs to be padded to a predetermined number of characters. This results in each row of the DAPHNE matrix having a predetermined size in bytes. This is necessary, as the distributed worker calculates an offset given a starting row when writing/reading to/from a Lustre file. Alternative ways to calculate the byte offset each worker should operate on would inevitably introduce the additional complexity of translating byte offsets to rows and vice versa. When handling serialized DAPHNE objects, no padding is necessary since we are dealing with raw data, meaning their size in bytes is predetermined. Apart from this padding, the Lustre Read and Write kernels for the distributed runtime are very similar in logic to the ones using the local filesystem.

Lustre being POSIX-compliant means we can use an offset to read/write at a specific location. This allows a distributed design where each worker handles a different segment of the matrix whilst performing operations to the same file. The segment of the file that each worker is responsible for can be calculated given a starting row of the matrix, hence the requirement for the row size to be set. For now, only synchronous gRPC is supported as the backend for the distributed runtime. Finally, we note that each file is independent of the method used to create it. This means that the same file can be used by a local runtime or a distributed runtime with varying numbers of workers transparently.

The code for our integration is currently on a separate repository, since there are some conflicts with the dependencies. For this reason, the current working version of the prototype builds upon an older DAPHNE commit (commit 336420e4b40e20a6ab43709d78c95edc61b231a7, 25 September 2024). We will soon investigate the best way to merge this integration upstream.

2.2 Kernel and Filetype Support

2.2.1 Kernel updates

Since Deliverable D4.3 [D4.3] we have further expanded DAPHNE's existing collection of C++-based runtime kernels. In the following, we give a short overview of the newly added kernels.

- **Sparse data**: We added a kernel specialization of matrix multiplication on CSR matrices.
- **Lists**: WP3 introduced a new list data type in DaphneDSL for storing matrices of homogeneous data/value type and heterogeneous shapes. The DAPHNE runtime offers kernels for operations on these lists such as `createList`, `append`, and `remove`.
- **Deep neural networks (DNN)**: So far, the DAPHNE runtime provided CUDA-based kernels for typical DNN operations such as convolution, max/avg-pooling, or bias addition, targeting the execution on GPUs (in collaboration with WP7). As not all DAPHNE users have a GPU set up, we decided to add fallback kernels for CPU execution for all DNN-related operations supported by DAPHNE.
- **String handling**: In collaboration with WP3, we added support for a string value type in DAPHNE matrices and frames, i.e., strings are supported as the elements of these data structures. To this end, we adapted various existing kernels such that they can be instantiated for the string value type. This includes elementwise comparisons, string manipulation operations (elementwise lower/upper case, concatenation, length), casts, reorganization (transpose, reverse, order), and left/right indexing.
- **Data preprocessing**: We added kernels for essential data preprocessing operations such as recoding and binning.
- **Control flow**: We added a kernel for the new `stop()` built-in function in DaphneDSL, which allows DAPHNE users to terminate the program execution, e.g., in case application-specific constraints are violated.
- **CUDA-based GPU kernels**: In collaboration with WP7, we implemented additional elementwise binary operations, such as `min`, `max`, and `not-equal`.
- **Miscellaneous kernels**: Furthermore, we added support for additional elementwise unary operations (e.g., trigonometric functions, logarithm, isNan) and additional full/row-wise/column-wise aggregations (e.g., mean, variance, standard deviation).

With these additional kernels, DAPHNE can be used more smoothly and productively. In fact, many of these kernels were added in reaction to the needs of our use case partners from WP8.

2.2.2 Filetype support

The **Coordinate List (COO)** sparse matrix representation was integrated into the DAPHNE runtime, enhancing its ability to handle sparse matrices efficiently. The COO format, unlike traditional dense or CSR (Compressed Sparse Row) formats, is especially suited for ultra-sparse matrices by storing only the non-zero values along with their row and column indices, thus optimizing memory usage. The implementation involved creating a `COOMatrix` class with attributes to manage matrix values, row, and column indices. Key methods include:

- Getters and Setters: For accessing and modifying matrix elements.
- Aggregation Kernels: New aggregation methods to handle operations across rows, columns, or the entire matrix.
- Element-wise Operations: Support for unary and binary element-wise operations on COO matrices.
- Random Matrix Generation and Transposition: For efficiently generating random COO matrices and performing transpositions.

The **TDMS (Technical Data Management Streaming)** file format was also integrated into the DAPHNE project, enhancing its data processing capabilities. TDMS is widely used in engineering and manufacturing for high-performance data acquisition, often handling large-scale datasets. This supports DAPHNE's goal of robust data processing, enabling it to handle more complex, large-scale datasets efficiently, particularly in technical and engineering domains. Support for TDMS was of particular interest to our use-case partners (KAI – WP8).

TDMS files are organized in a three-level hierarchy (file, group, and channel) to manage large datasets effectively. Each file contains groups, which contain channels where the actual data is stored. Channels store the raw data, while metadata provides structure. TDMS files are composed of segments with three parts—Lead In, Metadata, and Raw Data. This layout allows for efficient data storage and access. Segments store metadata on the file structure and indexing, enabling quick retrieval of specific data segments.

For implementing TDMS reader methods, a C++ library[8] to read TDMS files into DAPHNE’s structures (DenseMatrix and Frame) was utilized. This involved parsing TDMS data and mapping it to DAPHNE matrices or frames, organizing data by group and channel.

For TDMS writer methods: The writers convert DAPHNE matrices and frames into TDMS files by assigning a single group with multiple channels (one for each matrix column). Each data column is assigned a unique channel to facilitate structured data storage. A helper library[9] is used for generating files in the TDMS format.

2.3 NUMA-awareness and Data Locality

NUMA-aware data placement is an optimization technique used in systems with Non-Uniform Memory Access (NUMA) architectures. In NUMA systems, memory is divided among different nodes (typically CPU sockets). Each node exhibits faster access time to its own local memory. Accessing memory located on a different node incurs higher latency and lower bandwidth. NUMA-aware data placement thus organizes data in memory so that it is located as close as possible to the CPU or thread that will use it. This minimizes remote memory accesses,

[8] https://github.com/gmarkantonatos/TDMS-reader based on https://www.iondev.ro/tdms/
[9] https://github.com/MahdaSystem/TDMS/tree/master

improving data access speed and overall system performance. In terms of how memory is allocated (i.e., Memory Allocation Policies) we can briefly define the following NUMA policies:

- *Preferred Node:* Memory is allocated from a preferred node, but if that node's memory is exhausted, other nodes are used.
- *Interleaved Allocation:* Memory is spread evenly across nodes, balancing memory load and bandwidth but potentially increasing access latency.
- *Local Allocation:* Memory is allocated on the same node as the CPU that requested it, which helps minimize remote memory access.

In DAPHNE, NUMA-aware efforts were two-fold: The *DAPHNE runtime scheduling* incorporates NUMA-aware strategies that enhance data locality and minimize memory access latency. The DAPHNE local scheduler handles tasks efficiently with multiple queue types: a centralized queue for all devices (and their NUMA domains) on the node, partially decentralized per-device group (i.e., a single NUMA domain) queues tailored, and per-device queues (i.e., NUMA domains on a device) that assign tasks to specific devices (e.g., CPUs or GPUs). Per-device group queues enable tasks to be placed within the same NUMA domain, reducing inter-domain data transfers and leveraging faster, local memory access. These queues are complemented by work assignment strategies that dynamically distribute tasks, either by work-sharing from centralized queues, work-stealing across distributed queues to balance load while maintaining locality, or a combination of both for queues organized hierarchically. For a comprehensive discussion of these queue types and work assignment strategies, refer to Deliverable D5.4 (under concurrent submission this period).

In terms of OS-level efforts, we have taken advantage of the fact that many operating systems, like Linux, provide options to control memory allocation policies to make applications NUMA-aware. Our study examines DAPHNE's matrix processing capabilities in NUMA systems, focusing on performance across matrix types and configurations on two systems: an Intel(R) Xeon(R) Gold and a QEMU virtualized environment, both over an AMD EPYC server. Through various NUMA policies and memory placements, experiments were conducted using Linux NUMA tools, such as `numactl` and `numastat`, for both sparse and dense matrices. Scenarios considered include cases where allocated memory fits within a single NUMA node and cases requiring memory to span multiple NUMA nodes.

For the evaluation, we use the DAPHNE implementation of the PageRank and Connected Components algorithms, available on the DAPHNE Github repository. Both algorithms use an adjacency matrix to store the input graph and feature a tight loop of linear algebra operations (matrix-vector and elementwise multiplications). For the input matrix, we synthetically generate (via uniform sampling) a ~18GiB matrix, which we instantiate within the DAPHNE runtime as either dense or sparse (CSR) matrix and evaluate the effect NUMA policies for both cases.

2.3.1 Experiments with Linux NUMA policies

2.3.1.1 System 1: Intel Xeon Gold

We first show the results of various NUMA configurations on a 2-socket Intel Xeon Gold (Skylake) server with 128GiB of DDR4 memory per socket. For this set of experiments, the memory allocated by the DAPHNE runtime fits entirely within a single NUMA node. The results are normalized to the performance of the baseline default Linux scenario, i.e., when we run DAPHNE without tweaking any Linux NUMA policies or configuration.

PageRank Algorithm

- **Sparse Matrix Performance**: For PageRank with sparse matrices on the Xeon Gold system, where the dataset fit entirely within a single NUMA node, performance remained stable or worsened across NUMA configurations. Minor performance gains were observed under `--interleave=all` and `–m0` policies, but the default Linux policy delivered nearly optimal performance. This stability suggests that, for sparse matrices on this system, PageRank generally benefits from NUMA policies, given the dataset's confinement to a single NUMA node.
- **Dense Matrix Performance**: Dense matrices, in contrast, showed a marked response to NUMA configurations. The `–m0` policy provided substantial performance improvements, likely due to better alignment of memory with CPU locality, which is beneficial for dense data. This reduction in access latency highlights the importance of specific NUMA-aware optimizations for dense matrices within a single NUMA node, where efficient memory-thread alignment can reduce runtime.

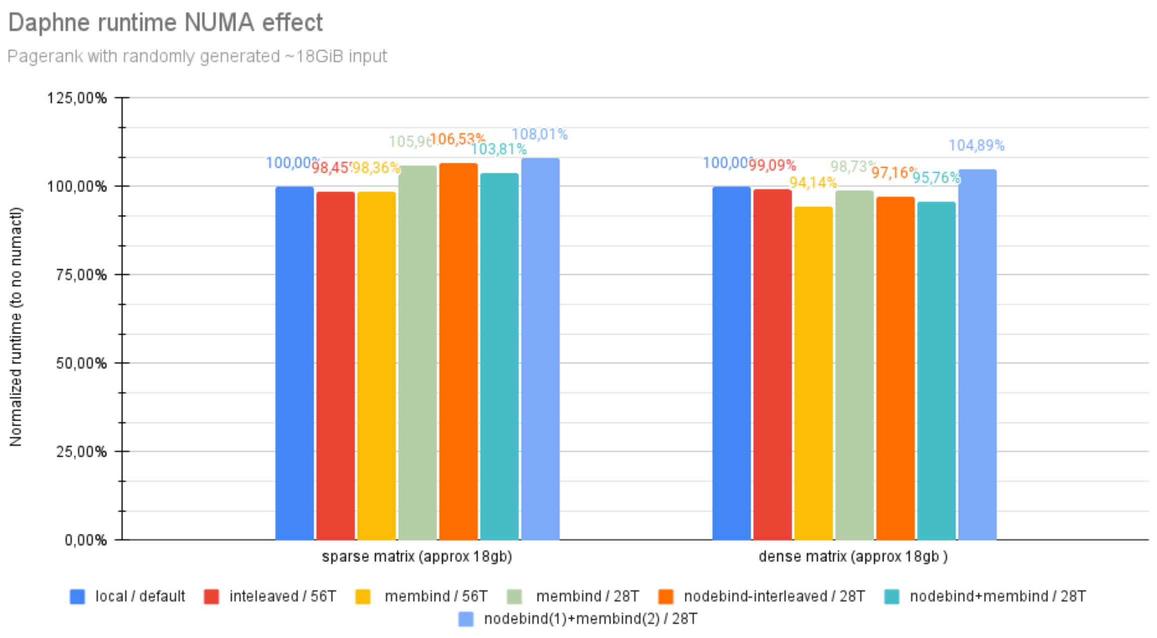

Figure 4: PageRank on Xeon Gold

Connected Components Algorithm

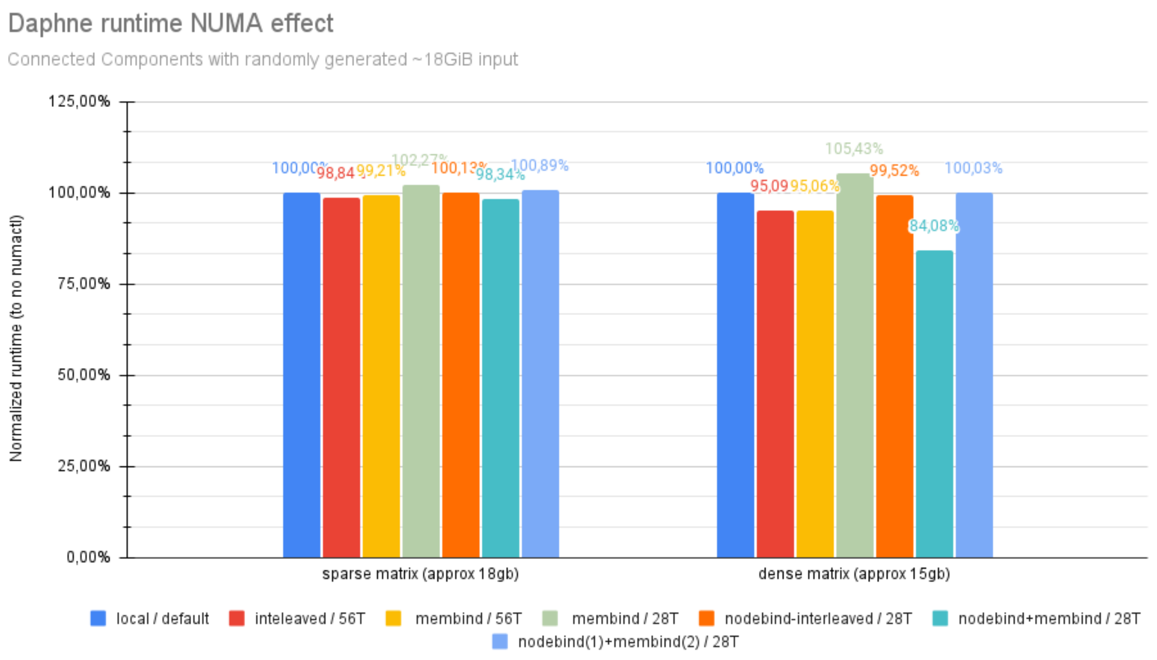

Figure 5: Connected Components on Xeon Gold

- **Sparse Matrix Performance**: When executing the connected components algorithm on sparse matrices, the Xeon Gold server showed similar stability across NUMA configurations as observed with PageRank. Alternative policies offered negligible improvement, and in some cases, deviated slightly from default performance. This consistency indicates low sensitivity of sparse matrix computations to NUMA configurations when datasets fit within a single NUMA node.
- **Dense Matrix Performance**: Dense matrices showed substantial gains when the `-N0 -m0` policy was applied, achieving a 15% improvement over other configurations. This performance boost likely stems from `-N0 -m0` directing both memory allocation (`-m0`) and thread execution (`-N0`) to the same NUMA node, optimizing access times by reducing cross-node latency. Such alignment enhances data locality, crucial for dense matrix workloads where frequent memory accesses benefit from minimized NUMA overhead. These results underscore the potential of localized NUMA policies like `-N0 -m0` to enhance performance in dense matrix operations, especially in algorithms like connected components that are sensitive to memory access latency.

2.3.1.2 System 2: QEMU Virtualized Environment

For this set of experiments, we use QEMU on an AMD EPYC server to create a 2-socket virtual machine with smaller-sized NUMA nodes. This way, we evaluate the effect of various NUMA configurations and policies on DAPHNE performance, when the input data cannot fit in a single NUMA node. In similar fashion to the previous section, we use a uniformly-sampled synthetic input matrix, and we report the results for various NUMA configurations for both dense and sparse matrices for the Page Rank and Connected Components algorithms, normalized to the performance of these algorithms when using the default Linux NUMA policy.

PageRank Algorithm

- **Sparse Matrix Performance**: Sparse matrices running PageRank on the QEMU environment (which required memory to span two NUMA nodes) showed significant performance improvement with the `--interleave=all` configuration, achieving approximately 7.5% improved efficiency. `Numastat` indicated a reduction in NUMA misses with interleaving, reinforcing its effectiveness in multi-node memory scenarios. Conversely, executing in a single node resulted in performance degradation due to memory contention, highlighting the need for interleaving in NUMA-limited environments for sparse matrix PageRank.
- **Dense Matrix Performance**: For dense matrices, the `--interleave=all` policy again provided improved results on QEMU, as this configuration helped distribute

memory across nodes more evenly, reducing contention and improving runtime. Dense matrices benefit more noticeably from interleaving under multi-node conditions, where memory spanning and workload balancing are critical to performance.

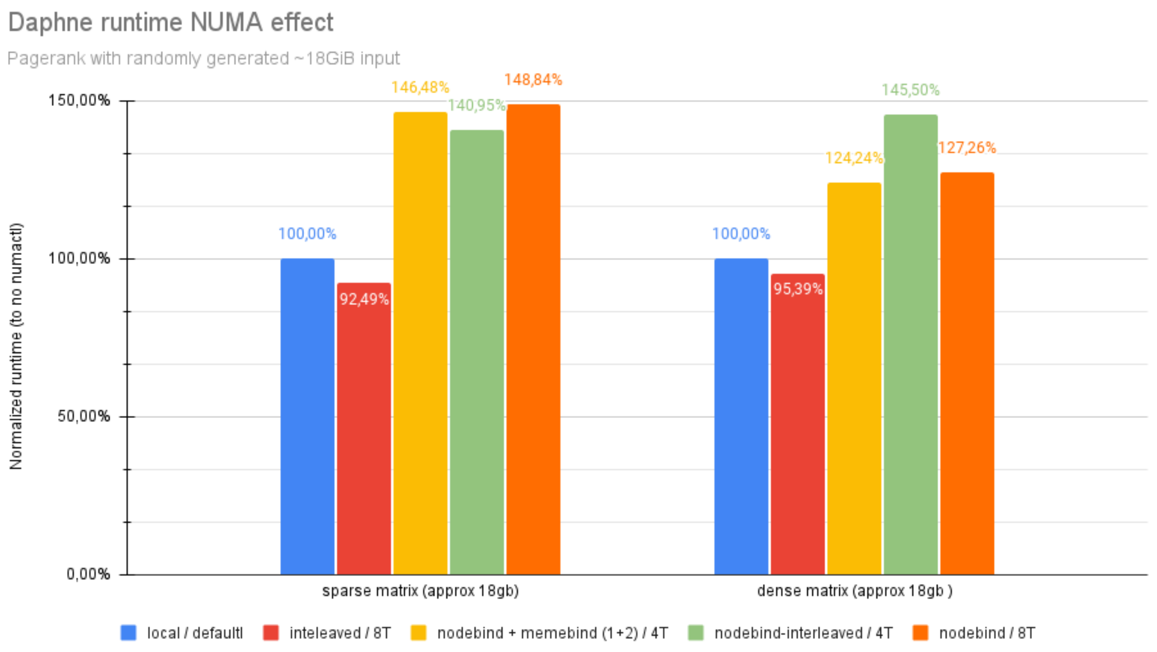

Figure 6: PageRank on QEMU system

Connected Components Algorithm

- **Sparse Matrix Performance**: Sparse matrices in the connected components algorithm on QEMU performed best under the default NUMA policy. The default configuration appears to optimize sparse memory access patterns sufficiently, reducing potential NUMA misses without requiring interleaving. This result indicates that sparse matrices in the connected components algorithm can achieve efficient memory access with default policies, avoiding the additional overhead interleaving can introduce in certain multi-node scenarios.
- **Dense Matrix Performance**: Dense matrices in connected components also showed substantial improvements with `--interleave=all`, achieving around 10% better performance than the default policy. This improvement highlights the advantages of interleaving for dense matrices, as it minimizes NUMA-related latency and ensures more consistent memory access patterns. By mitigating NUMA overhead, interleaving can notably benefit dense datasets spanning multiple NUMA nodes.

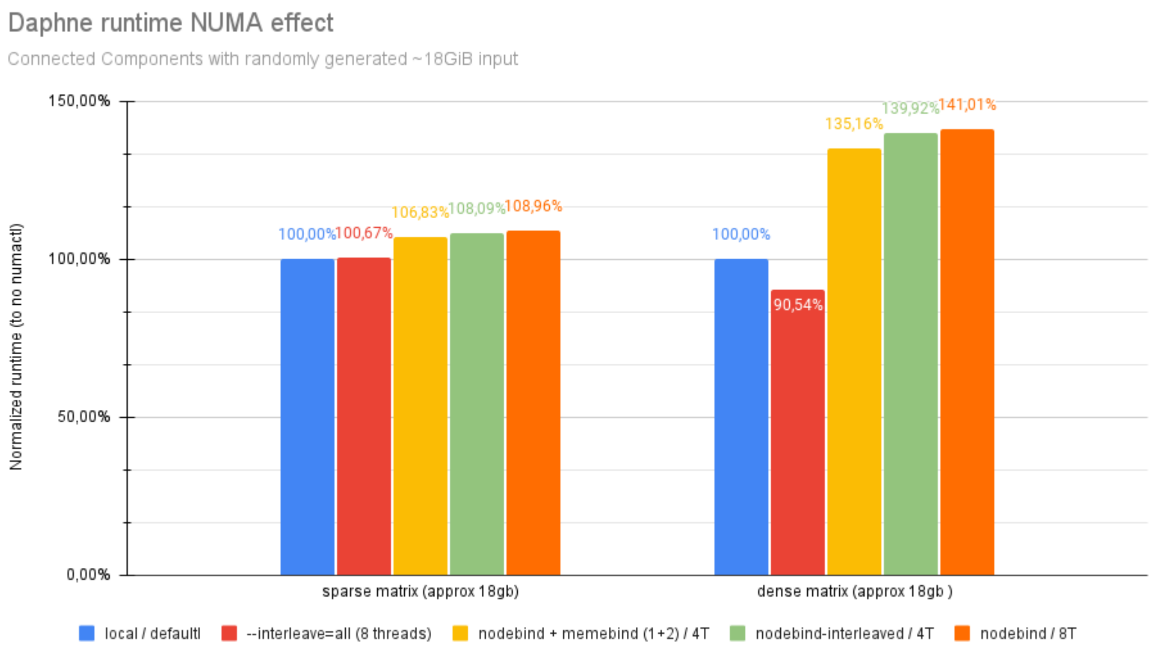

Figure 7: Connected Components on QEMU system

2.3.1.3 Discussion

Across both systems, sparse matrices displayed relatively stable performance with minimal sensitivity to NUMA configurations, particularly when confined to a single NUMA node. Dense matrices, on the other hand, demonstrated a greater dependency on NUMA-aware configurations, with `--interleave=all` and `-m0` yielding noticeable gains in multi-node environments. For both PageRank and Connected Components, interleaving was beneficial for dense data on QEMU, where memory spanning across nodes introduced potential contention. The notable 15% performance gain observed with `-N0 -m0` for Xeon Gold further emphasizes the importance of aligning memory and thread locality in NUMA-aware systems. However, sparse matrices running Connected Components on QEMU benefitted most from the default policy, suggesting that interleaving can sometimes introduce unnecessary overhead for certain algorithms and matrix types.

2.3.2 Impact of NUMA Policies and Queue Layouts

The purpose of these experiments is to evaluate the performance of the DAPHNE framework's task scheduling and memory access strategies under different configurations, especially when dataset sizes exceed the memory capacity of a single NUMA node. When datasets span across multiple NUMA nodes, performance can be influenced by both memory access patterns

(NUMA policies) and task distribution methods (queue layouts). Understanding these interactions is critical in NUMA environments, where efficient memory locality and task distribution play a significant role in minimizing latency and optimizing computational throughput.

In these tests, we ran the PageRank and Connected Components algorithms with various queue layouts and NUMA configurations to assess how these factors affect performance in the context of distributed memory. We used the same virtualized (QEMU) setup as described before. Specifically, we experimented with both the default allocation and interleave policy in combination with three queue layouts: `CENTRALIZED, PERGROUP,` and `PERCPU`. The queue layouts represent different approaches to work assignment in the DAPHNE framework, with `CENTRALIZED` relying on a single queue for all tasks, while `PERGROUP` and `PERCPU` introduce multiple work queues to support work-stealing mechanisms. These layouts help distribute tasks across CPUs to reduce bottlenecks and optimize core utilization, depending on whether task queues are assigned per worker (PERCPU) or per group of workers (PERGROUP). The resulting plots display the runtime across various NUMA and queue layout configurations, highlighting the impact of these setups on algorithm performance.

These experiments will reveal how different NUMA policies and queue layouts impact computational performance when data must be spread across multiple nodes, rather than residing in one. By comparing default and interleaved memory policies with centralized and distributed work queues, this analysis aims to identify configurations that balance task distribution with memory access efficiency, ultimately guiding optimization strategies for the DAPHNE framework in NUMA-based, multi-node environments. In these comparisons, the same physical size (~18 GiB) is used for both sparse and dense datasets, which explains why sparse operations appear slower than dense ones. This explicit choice ensures a fair evaluation of memory policies and queue layouts under consistent physical memory constraints.

2.3.2.1 PageRank Analysis

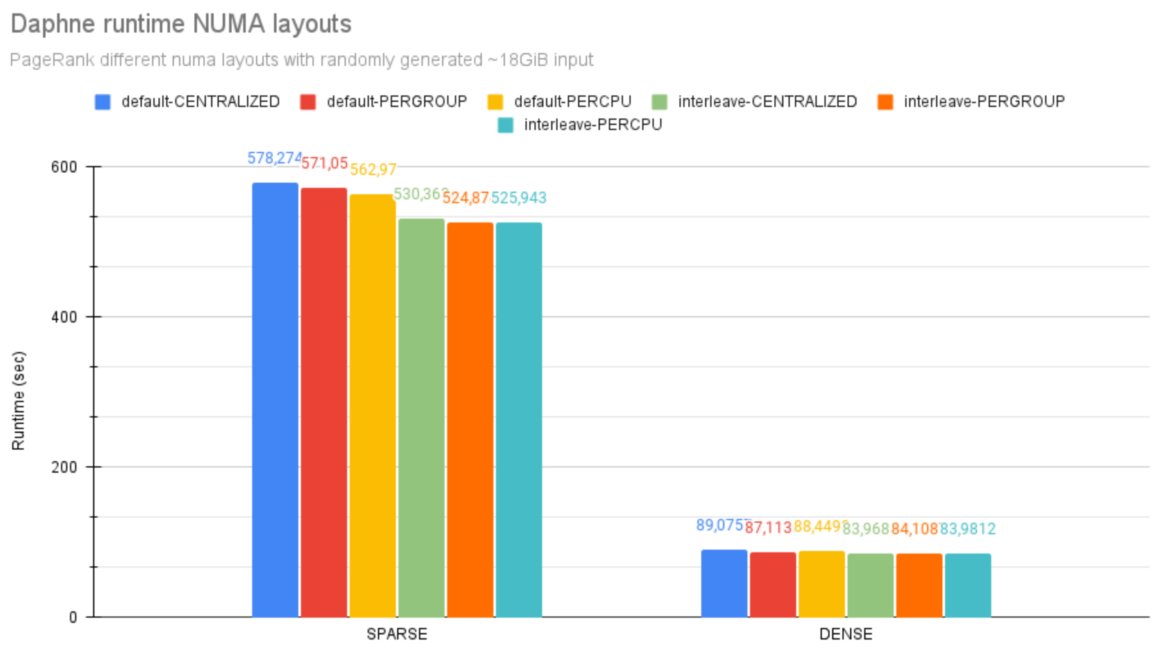

Figure 8: PageRank for different NUMA Layouts

- **Sparse Matrix Performance:** In the sparse matrix case, default-CENTRALIZED showed the slowest performance, which is consistent. The centralized queue layout can lead to task contention and inefficient memory access patterns, particularly in a NUMA environment where a single queue creates memory access bottlenecks across nodes. This layout also increases cross-node memory access as all CPUs compete for tasks from the same queue, adding latency. Among default executions, *default-PERCPU* had the best performance, with *default-PERGROUP* also showing improvement over centralized. *PERCPU's dedicated queue for each CPU* allowed better task distribution and minimized contention, improving access locality for CPUs accessing local memory. *PERGROUP*, which organizes tasks by node, helped maintain more efficient memory access patterns, reducing the amount of cross-node access compared to CENTRALIZED. With interleaving, memory is spread across nodes, balancing memory demands and improving access for all layouts. All interleave configurations performed significantly better than their default counterparts, as the memory load was distributed across both nodes. The *interleave-PERGROUP layout* had the best performance, though it was very close to *interleave-PERCPU*. This suggests that both layouts benefit similarly under interleaving, with *PERGROUP* having a slight edge due to its alignment with NUMA nodes, reducing cross-node latency while balancing tasks effectively. *Interleave-CENTRALIZED* also performed better than any default configuration, showing that even a centralized queue benefits from distributed memory access across nodes when interleaving is enabled.

- **Dense Matrix Performance:** For dense matrices, the performance differences across queue layouts in default mode were minimal. *Default-PERGROUP* showed the best performance, though only slightly better than *PERCPU* and *CENTRALIZED*. Dense matrix operations require continuous memory access, and *default-PERGROUP's* node-based task grouping aligns well with this requirement, allowing CPUs in each node to access memory without excessive cross-node calls. The small performance differences across default queue layouts suggest that, for dense matrices, the locality of memory access is already well-managed even without strict queue isolation. With interleaving enabled, all layouts saw improved performance, as memory is spread evenly across nodes, preventing any single node from becoming a bottleneck. *Interleave-PERCPU* achieved the best performance, although *interleave-PERGROUP* and *interleave-CENTRALIZED* were very close. This slight edge of *PERCPU* under interleaving could be due to each CPU's dedicated queue, which helps distribute tasks while maintaining consistent access to interleaved memory. However, the differences are minimal, indicating that interleaving has a more substantial impact on dense matrix operations than the specific queue layout.

2.3.2.2 Connected Components Analysis

- **Sparse Matrix Performance:** In the sparse matrix case, the *default-CENTRALIZED layout* showed the worst performance among the queue layouts. This result aligns with expectations, as a centralized queue can lead to congestion, especially when dealing with large datasets that require frequent memory access. With all workers fetching tasks from a single queue, contention increases, which can be particularly problematic in a NUMA environment. This layout may also lead to more random memory access patterns as CPUs on different nodes pull from a single queue, resulting in less efficient memory locality. *Default-PERGROUP* and *default-PERCPU* layouts showed improvements over centralized. The *default-PERCPU* layout achieved the best performance among the default executions. Since each CPU has its own dedicated queue, PERCPU minimizes task contention and better utilizes local memory access, which reduces latency and cross-node memory access for NUMA systems. By isolating each CPU's work queue, PERCPU enables workers to maintain consistent access to local memory, leading to fewer interruptions and more efficient task execution for a large, distributed sparse matrix. When *interleave* was applied, all queue layouts (*CENTRALIZED, PERGROUP,* and *PERCPU)* performed similarly, with *interleave-PERCPU* showing a slight edge as the best performer. Interleaving spread memory usage across both NUMA nodes, balancing memory load across nodes and reducing the risk of any single node's memory becoming overloaded. The fact that interleave-PERCPU, interleave-PERGROUP, and interleave-CENTRALIZED achieved similar performance suggests that, with memory interleaving, the choice of queue layout has less impact on performance. The memory demands were effectively distributed across nodes, which reduced contention and allowed all layouts to achieve near-optimal performance. However, interleave-PERCPU's dedicated queues for each CPU likely contributed to its slight advantage by keeping memory access consistently aligned with each CPU's work.

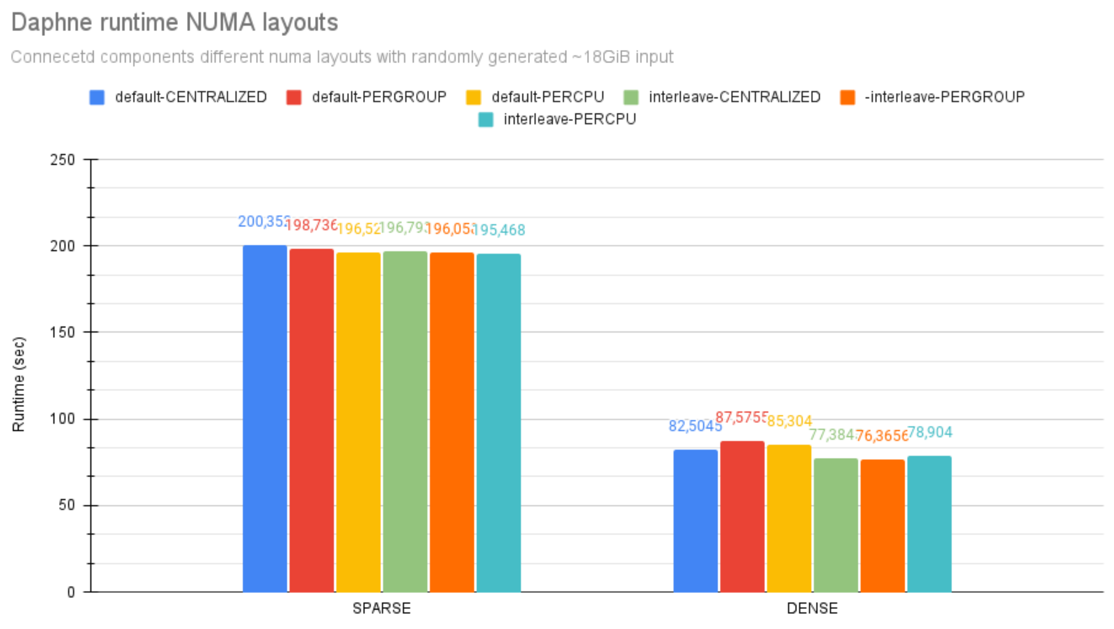

Figure 9: Connected Components with different NUMA Layouts

- **Dense Matrix Performance:** For the dense matrix, *default-CENTRALIZED* showed the best performance among the default queue layouts. Dense matrices require more continuous memory access, and a centralized queue can be beneficial because it simplifies task scheduling and reduces the complexity of managing tasks across multiple queues. With all workers pulling from a single queue, memory locality is less fragmented, which likely contributed to default-CENTRALIZED's better performance. In contrast, *default-PERGROUP* and *default-PERCPU* showed slightly slower performance, likely due to the increased overhead of managing multiple queues and more complex memory access patterns as tasks are distributed more broadly. However, the interleave policy improved performance across all queue layouts for the dense matrix, with *interleave-PERGROUP* achieving the best performance. With interleaving, memory is spread across both NUMA nodes, which reduces the chance of memory bottlenecks and ensures a balanced memory load. The interleave-PERGROUP layout, in particular, benefited from this setup by combining the memory load-balancing of interleaving with task distribution across NUMA nodes. This setup likely allowed PERGROUP to minimize latency, as each group of CPUs could access memory efficiently across nodes without overloading any single node.

2.3.3 Summary of Observations

These experiments provided valuable insights into the impact of NUMA policies and queue layouts on the performance of the PageRank and Connected Components algorithms within a QEMU virtualized environment with two NUMA nodes. By testing different NUMA configurations (default and interleave combined with CENTRALIZED, PERGROUP, and PERCPU queue layouts) we observed how task distribution and memory access strategies affect execution times for large sparse and dense matrices.

For both algorithms, default-CENTRALIZED demonstrated the poorest performance in most cases, highlighting the limitations of a single, centralized work queue when handling large datasets across multiple NUMA nodes. The CENTRALIZED layout led to higher contention and cross-node memory access, which contributed to bottlenecks, particularly for sparse matrices where task and memory demands are more distributed.

PERCPU and PERGROUP layouts, however, generally improved performance in default mode by reducing task contention and aligning memory access more closely with the executing CPUs. Default-PERCPU performed particularly well with sparse matrices, benefiting from dedicated queues that helped minimize cross-node access and improved locality for tasks within each CPU.

Interleaving memory across NUMA nodes proved advantageous across all configurations, balancing memory load and reducing single-node memory congestion. Interleaved configurations consistently outperformed their default counterparts for both algorithms and matrix types, with interleave-PERGROUP and interleave-PERCPU showing the best results. These layouts balanced task distribution effectively while leveraging interleaved memory, resulting in minimal cross-node latency and enhanced performance.

In summary, these experiments underscore the importance of aligning NUMA memory policies and queue layouts to the algorithm's memory access patterns and the data type. Interleaving provided significant improvements by distributing memory evenly across nodes, and PERGROUP and PERCPU layouts proved most effective overall. These findings indicate that, in NUMA environments, a combination of interleaving and task-distributed queue layouts can optimize performance for large-scale, memory-intensive computations in systems where data spans multiple nodes.

2.4 Other Runtime Enhancements

Additional features added in the runtime during this reporting period are the following:

- Time measurement of individual kernel calls to find the most time-consuming kernels in a DaphneDSL script execution (`--statistics` flag).
- More systematic and consistent error messages. Kernels are mapped to their source code location in a DaphneDSL script and can thus generate much more useful error messages (in cooperation with WP3).
- Efficient (zero-copy) data exchange between DaphneLib and popular Python libraries (numpy, pandas, TensorFlow and PyTorch).

3 Final DAPHNE Runtime Prototype

3.1 Artifact Access and Use

3.1.1 Introduction

The DAPHNE DSL runtime prototype described in this deliverable is publicly accessible as a snapshot of the DAPHNE development repository (created on November 9th, 2024) under the following link: https://daphne-eu.know-center.at/index.php/s/Wcc2KCG2gG9jKsE

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

In [D4.2] and [D4.3] we provided detailed guidelines on how to build DAPHNE, run a simple DSL script and execute pipelines using the distributed runtime with different distributed backends. The same steps can be repeated to build the latest version of DAPHNE provided in the current artifact. HDFS is fully integrated and can now be used with DAPHNE; however it is not installed by default. First, we need to install the required dependencies on the system. These are listed in the Getting Started[10] Guide on GitHub. Regarding HDFS, libhdfs3 requires these additional dependencies:

• boost (tested on 1.53+)	http://www.boost.org/
• google protobuf	http://code.google.com/p/protobuf/
• libxml2	http://www.xmlsoft.org/
• kerberos	http://web.mit.edu/kerberos/
• libuuid	http://sourceforge.net/projects/libuuid/
• libgsasl	http://www.gnu.org/software/gsasl/

Note that the DAPHNE container images already ship with all these dependencies installed. Finally, we need to re-build DAPHNE and provide the HDFS flag:

```
$ ./build.sh --hdfs
```

The DAPHNE docker containers contain all the necessary HDFS dependencies and make it easier to run or re-build DAPHNE.

In order for DAPHNE to utilize the HDFS file system, certain command line arguments need to be passed (or included in the configuration file).

- `--enable-hdfs`: A flag to enable hdfs.
- `--hdfs-address=<IP:PORT>`: The IP and port HDFS listens to.

[10]https://github.com/daphne-eu/daphne/blob/main/doc/GettingStarted.md

- `--hdfs-username=<username>`: The username used to connect to HDFS.

3.1.2 Reading from HDFS

In order to read a file from the HDFS, some pre-processing must be done. Assuming the file is named FILE_NAME, a user needs to upload the file into HDFS. DAPHNE expects the file to be located inside a directory with some specific naming conventions. The path can be any path under HDFS, however the file must be named with the following convention:

```
/path/to/hdfs/file/FILE_NAME.FILE_TYPE/FILE_NAME.FILE_TYPE_segment_1
```

FILE_TYPE is either .csv or .dbdf (DAPHNE binary data format) followed by .hdfs, e.g. myfile.csv.hdfs.

The suffix _segment_1 is necessary, since we support multiple writers at once (see more below), the writers need to write into different files (different segments). In this case where the user pre-uploads the file, it needs to be in the same format, but just one segment.

Each segment must also have its own .meta file within the HDFS. This is a JSON file containing information about the size of the segment as well as the type. For example myfile.csv.hdfs_segment_1.meta:

```
{
    "numCols": 10,
    "numRows": 10,
    "valueType": "f64"
}
```

We also need to create a .meta file containing information about the file, within the local file system (from where DAPHNE is invoked). Similar to any other file which will be read by DAPHNE, we need to create a .meta file, which is in JSON format, containing information about where the file is, information about the rows/cols etc. The file should be named: FILE_NAME.FILE_TYPE.meta, e.g. myfile.csv.hdfs.meta. The meta file should contain all the regular information any DAPHNE meta file contains, but in addition, it also contains information about whether this is an HDFS file and where it is located within HDFS:

```
{
    "hdfs": {
        "HDFSFilename": "/path/to/hdfs/file/FILE_NAME.FILE_TYPE",
        "isHDFS": true
    },
    "numCols": 10,
    "numRows": 10,
    "valueType": "f64"
}
```

Example:

Assume we have a dataset called training_data.csv which we want to upload to HDFS and use it with DAPHNE.

First, we need to upload the file under directory “datasets” and create the segment .meta file. HDFS should look like this:

```
$ hdfs dfs -ls /

/datasets/training_data.csv.hdfs/training_data.csv.hdfs_segment_1

/datasets/training_data.csv.hdfs/training_data.csv.hdfs_segment_1.meta

$ hdfs dfs -cat \

 /datasets/training_data.csv.hdfs/training_data.csv.hdfs_segment_1.meta

{"numCols":10,"numRows":10,"valueType":"f64"}
```

We also create the local .meta file:

```
$ cat ./training_data.csv.hdfs.meta

{"hdfs":{"HDFSFilename":"/datasets/training_data.csv.hdfs","isHDFS":
true},"numCols":10,"numRows":10,"valueType":"f64"}
```

Example DAPHNE script (code.daph):

```
X = readMatrix("training_data.csv.hdfs");

print(X);
```

Finally, we run DAPHNE providing the needed command line arguments:

```
./bin/daphne --enable-hdfs --hdfs-ip=<IP:PORT> --hdfs-username=ubuntu
code.daph
```

3.1.3 Writing to HDFS

In order to write to HDFS we just need to use the writeMatrix function like we would for any other file type and specify the hdfs suffix. For example:

1. DAPHNE script:

```
X = rand(10, 10, 0.0, 1.0, 1.0, 1);

writeMatrix(X, "randomSet.csv.hdfs");
```

2. Run DAPHNE:

```
./bin/daphne --enable-hdfs --hdfs-ip=<IP:PORT> --hdfs-username=ubuntu
code.daph
```

This will create the following files inside HDFS:

```
$ hdfs dfs -ls /

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1.meta


$ hdfs dfs -cat /randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1.meta

{"numCols":10,"numRows":10,"valueType":"f64"}
```

And also the .meta file within the local file system named randomSet.csv.hdfs.meta:

```
{
    "hdfs": {
        "HDFSFilename": "/randomSet.csv.hdfs",
        "isHDFS": true
    },
    "numCols": 10,
    "numRows": 10,
    "valueType": "f64"
}
```

3.1.4 Distributed Runtime and HDFS

Both read and write operations are supported by the distributed runtime.

3.1.4.1 Read

The exact same preprocessing must be performed, creating one file inside the HDFS with the appropriate naming conventions. Users can then run DAPHNE using the distributed runtime and depending on the generated pipeline, DAPHNE's distributed workers will read their corresponding part of the data speeding up IO significantly. Instead of the coordinator reading the data and then transmitting it to the distributed workers, each distributed node reads the specific part of the data it needs, based on the pipeline generated by the coordinator. Note that DAPHNE's compiler can be extended further to generate these pipelines optimally, assigning tasks depending on where data resides. For example:

DAPHNE script:

```
X = readMatrix("training_data.csv.hdfs");

print(X+X);
```

Run DAPHNE

```
$ export DISTRIBUTED_WORKERS=worker-1:<PORT>:worker-2:<PORT>
$ ./bin/daphne --distributed --dist_backend=sync-gRPC --enable-hdfs -
-hdfs-ip=<IP:PORT> --hdfs-username=ubuntu code.daph
```

3.1.4.2 Write

Similar to read, users just need to run DAPHNE using the distributed runtime flags. Notice that since we have multiple workers/writers, more than one segments are generated inside HDFS. The number of segments generated is always equal to the number of distributed workers used. Since we want to have a parallel write operation, different files (segments) need to be created so that each node can write to HDFS independently in parallel.

DAPHNE script:

```
X = rand(10, 10, 0.0, 1.0, 1.0, 1);

writeMatrix(X, "randomSet.csv.hdfs");
```

Run DAPHNE:

```
$ export DISTRIBUTED_WORKERS=worker-1:<PORT>:worker-2:<PORT>

$ ./bin/daphne --distributed --dist_backend=sync-gRPC --enable-hdfs -
-hdfs-ip=<IP:PORT> --hdfs-username=ubuntu code.daph
```

Assuming 2 distributed workers:

```
$ hdfs dfs -ls /

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1 #1st part of matrix

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1.meta

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_2 #2nd part of matrix

/randomSet.csv.hdfs/randomSet.csv.hdfs_segment_2.meta


$ hdfs dfs -cat /randomSet.csv.hdfs/randomSet.csv.hdfs_segment_1.meta

{"numCols":10,"numRows":5,"valueType":"f64"}

$ hdfs dfs -cat /randomSet.csv.hdfs/randomSet.csv.hdfs_segment_2.meta

{"numCols":10,"numRows":5,"valueType":"f64"}
```

And also the .meta file within the local file system named randomSet.csv.hdfs.meta.

3.1.5 Limitations

There are certain limitations when using HDFS with DAPHNE:

- Writing to a specific directory, through DAPHNE, within HDFS is not supported. DAPHNE will always try to write under the root HDFS directory /<name>.<type>.hdfs.

- Only synchronous gRPC is supported as the distributed backend for HDFS distributed read and write operations.

3.2 Evaluation Results

3.2.1 HDFS Integration

3.2.1.1 Setup

We deployed HDFS on our cluster, which consists of one coordinator and eight worker nodes, each equipped with 8 CPUs. The system was tested with both read and write operations to HDFS using multiple CSVs files, ranging from 86M, up to 2.6G in size. Each set of tests was repeated twice, comparing the impact of the replication factor from three to one. To ensure accuracy, each test was conducted five times, and the numbers presented in the plots below represent the average of these five trials. The results showed that read performance consistently improves when utilizing HDFS, taking full advantage of its distributed architecture for efficient data access. Additionally, write performance improves as more nodes are added to the system, demonstrating the scalability of HDFS when handling large datasets in distributed environments.

The results can be reproduced by following two steps:

1. Generate the CSV files: We used DAPHNE's randMatrix operation to generate the datasets presented in the result section:

```
A = rand($R, $C, 0.0, 1.0, 1.0, -1);
writeMatrix(A, $G);
```

2. Measure HDFS performance: We used the following DAPHNE code to measure the time it takes to read and write these CSV files both in the local FS and HDFS, using the setups described above:

```
t1 = now();
a = readMatrix($G);
t2 = now();
t3 = now();
// print(sum(a));
writeMatrix(a, $G);
t4=now();
// Print elapsed times in seconds.
print("read time[s]: ", 0, 0);
print((t2 - t1)*10.0^(-9));
print("write time[s]: ", 0, 0);
print((t4 - t3)*10.0^(-9));
```

3.2.1.2 Results

In the following Figures, we plot the time in seconds it takes for a Read & Write operation using:

a) The local filesystem (before HDFS integration),
b) a single-node HDFS, and
c) two to eight HDFS worker nodes,

for varying file sizes. In each line, we show two Figures with the performance of Read and Write operation over the same size CSV.

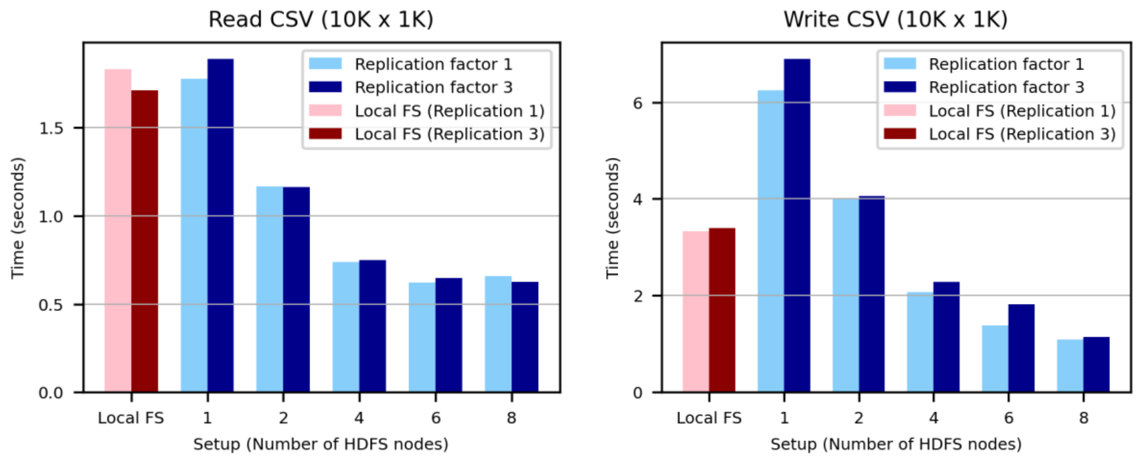

Figure 10: Read – Write HDFS performance for a 10K x 1K CSV file

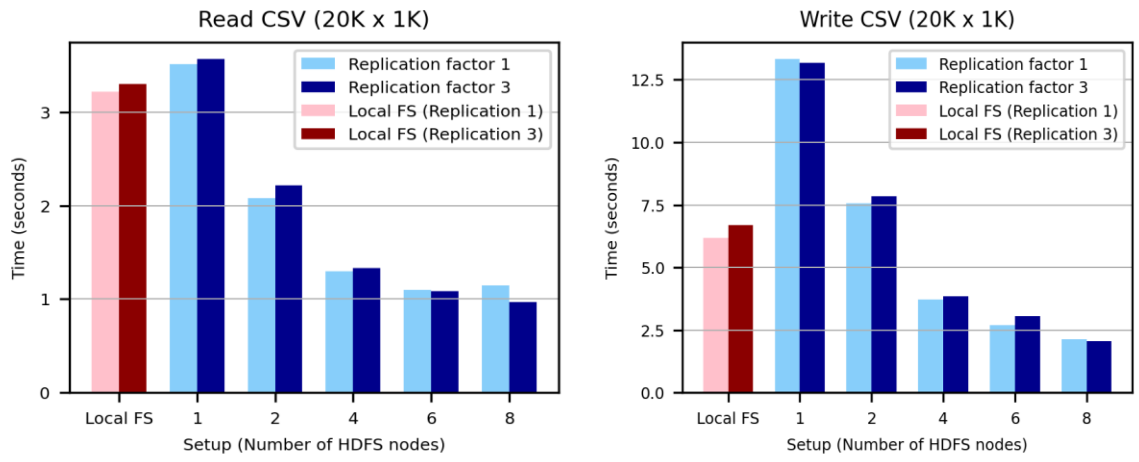

Figure 11: Read – Write HDFS performance for a 20K x 1K CSV file

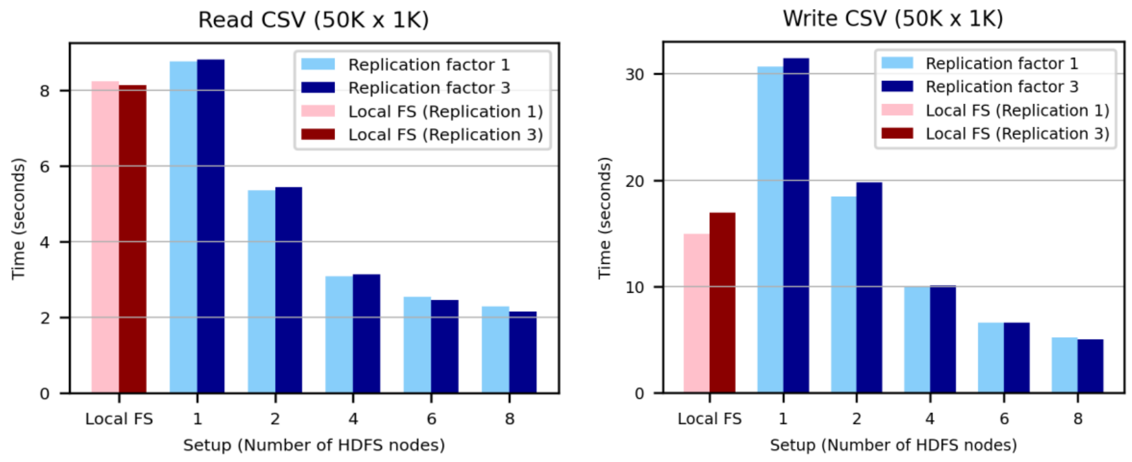

Figure 12: Read – Write HDFS performance for a 50K x 1K CSV file

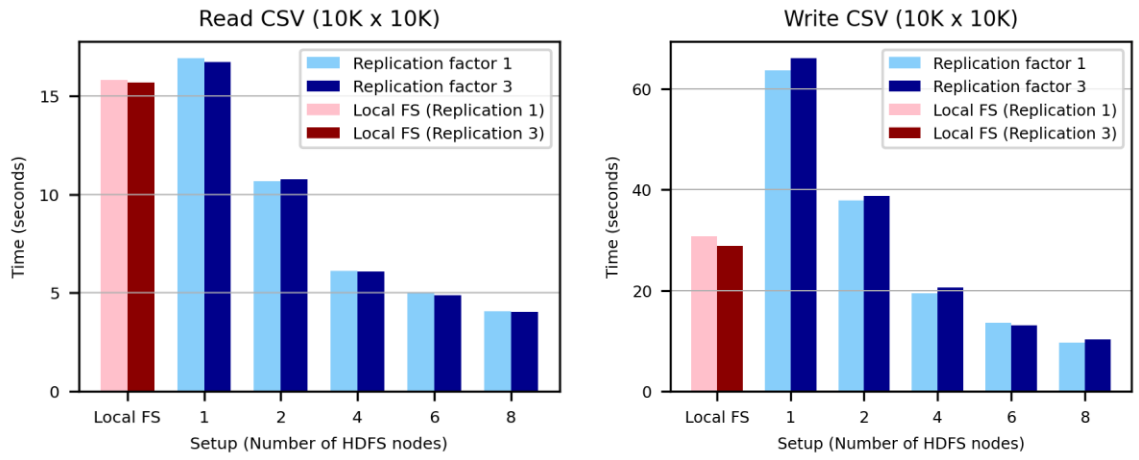

Figure 13: Read – Write HDFS performance for a 10K x 10K CSV file

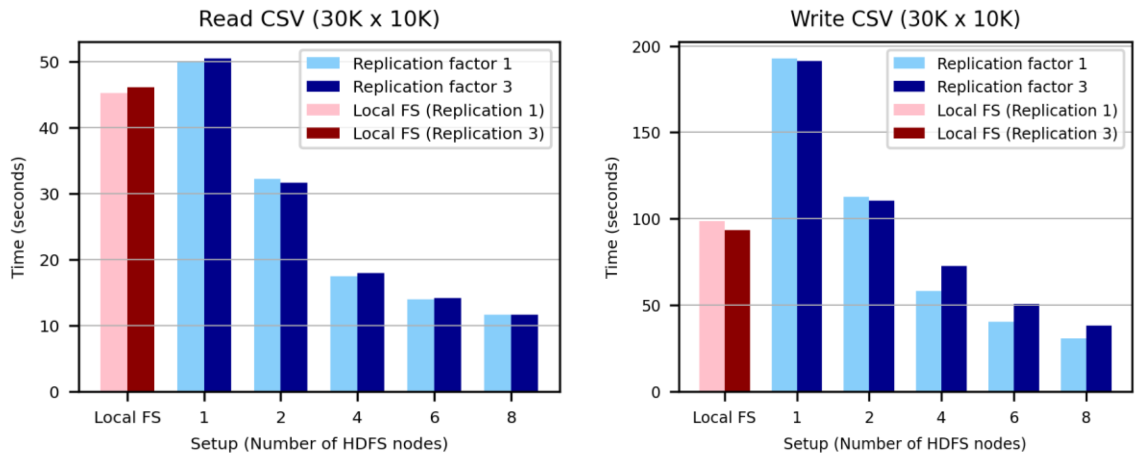

Figure 14: Read – Write HDFS performance for a 30K x 10K CSV file

The final Figure (Figure 15) demonstrates the effect of file size (for the largest cluster size of 8 workers) over the performance of our implemented Read and Write operations. The results confirm the following facts:

- HDFS scales gracefully as data size increases.
- Writing imposes bigger overhead that read operations.
- The replication factor has a minimal effect in case of VM co-location.

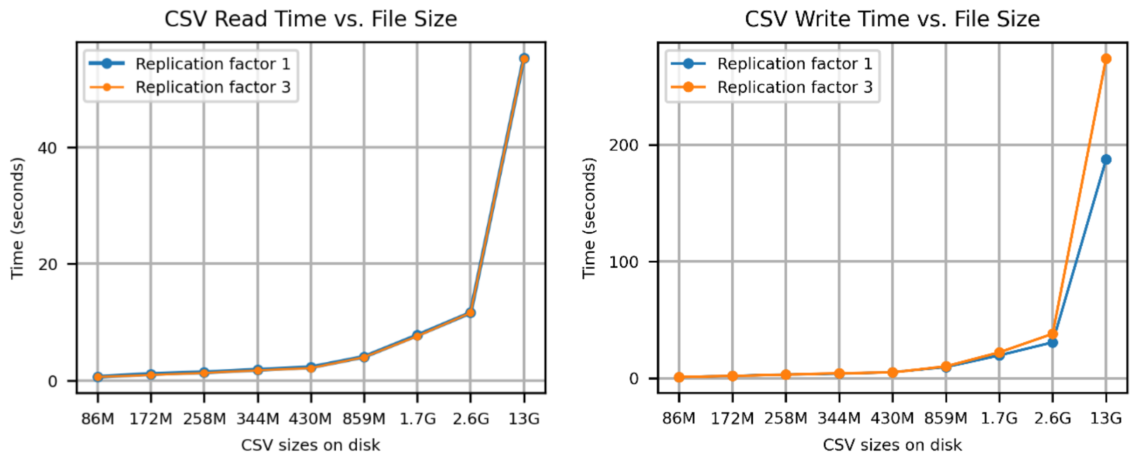

Figure 15: Read – Write HDFS performance vs. File size (8 worker nodes)

3.2.1.3 Discussion on Results

The results demonstrate a consistent improvement with the addition of more nodes. As the number of nodes increases from 2 to 8, both read and write times generally decrease, especially for larger matrix sizes. This indicates improved parallelization and resource utilization in a distributed environment, where more nodes allow for better distribution of tasks.

However, when examining the HDFS performance without distribution (HDFS filesystem on the coordinator node only), it is evident that execution times are consistently higher than those in the distributed HDFS setup. This highlights the clear benefits of leveraging a distributed computing approach.

That said, the results also reveal diminishing returns with higher node counts. While adding more nodes (up to 4 nodes) results in significant performance gains, the benefits start to taper off with 6 and 8 nodes for certain matrix sizes. This suggests that communication overhead or other factors, such as synchronization costs, might limit scalability beyond a certain point.

A notable observation is that write operations are slower than read operations across all tests. This is especially true for larger matrix sizes, such as 20000x1000 or larger. The difference in performance between read and write operations becomes even more pronounced in the distributed environment, indicating that write operations may face additional challenges in such setups.

Additionally, matrix size significantly impacts performance. Larger matrices, such as 50000x1000, exhibit more noticeable improvements in performance with distributed HDFS compared to smaller matrices. This suggests that larger workloads benefit more from the increased distribution and parallelization provided by additional nodes.

We also observe that the replication factor has minimal impact on performance overall, except when handling large datasets. For these larger CSV files, using a replication factor of one slightly improves write performance, as expected, since less data needs to be replicated across the cluster. Although read operations would typically benefit from a higher replication factor—due to the increased likelihood of local data availability for each node—in our setup, this advantage is diminished. Our cluster is composed of virtual machines hosted on a very small number of large physical servers which means that data locality is achieved regardless of the replication configuration.

Finally, a comparison between the local filesystem and distributed HDFS shows that, while the local filesystem performs adequately for smaller matrices, HDFS significantly outperforms it for larger matrices. This performance gap grows as more nodes are added.

3.2.2 Lustre Integration

3.2.2.1 Setup

Our experiments are conducted on an AWS-commissioned cluster, consisting of 9 t3.xlarge VMs. Each VM has 16GBs of RAM, 4 VCPUs, 2 EBS gp3 volumes: a 45GB one for the OS filesystem and an additional of 5GB for the Lustre filesystem (e.g., mounting OSTs). One VM plays the role of the DAPHNE coordinator and the Lustre MGS/MDT. The other eight VMs are DAPHNE workers and Lustre OSTs. All the VMs have to be Lustre clients to be able to access the Lustre filesystem.

The Lustre stripe size in all the experiments is 64 KB, which is the minimum value. For context, each system call via the Lustre kernels attempts to read/write up to 1048 KB. This means that system calls to the Lustre file system would include objects from all the OSTs. There is definitely room for experimentation here, as we expect the stripe size to be an important factor for the performance of the Lustre kernels. That being said, we keep the stripe size set in our experiments to reduce the number of variables.

When handling CSV files, our kernels need to pad the values to a fixed length. That length is currently hardcoded to be 17 characters. The actual values for the datatypes used can be represented accurately with a smaller number of characters (eight to be precise – generating equal sizes with the local filesystem). The difference in size between the files handled by Lustre and local kernels with two different padding sizes is displayed in Table 1. In the following experiments, we use the 17-characters-per-cell padding.

Table 1: Matrices and their respective sizes in Lustre experiments

Matrix Dimensions	Local FS kernel size ('.csv')	Lustre kernel size ('.csv.lustre') 17 chars per cell	Lustre kernel size ('.csv.lustre') 8 chars per cell
1000x5000	43 MB	43 MB	86 MB
1000x10000	86 MB	86 MB	172 MB
1000x20000	172 MB	172 MB	344 MB
1000x30000	258 MB	258 MB	515 MB
1000x40000	344 MB	344 MB	687 MB
1000x50000	430 MB	430 MB	859 MB
10000x10000	859 MB	859 MB	1,7 GB
10000x20000	1,7 GB	1,7 GB	3,4 GB
10000x30000	2,6 GB	2,6 GB	5,1 GB

3.2.2.2 Results

We perform the following simple experiment as a baseline and a proof of concept for the current Lustre kernels: A random matrix is generated and written to disk with varying size (number of rows and columns). The resulting file is read and written back to disk, using the corresponding kernel. The time for each operation is measured from within the DAPHNE script.

The DAPHNE script used to generate a csv file with R rows and C columns at location G is:

```
A = rand($R, $C, 0.0, 1.0, 1.0, -1);
writeMatrix(A, $G);
```

The DAPHNE script that reads the csv file from location G, writes it to location D and measures the elapsed time is:

```
t1 = now();
a = readMatrix($G);
t2 = now();
t3 = now();
```

```
writeMatrix(a, $D);
t4=now();
print("read time[s]: ", 0, 0);
print((t2 - t1)*10.0^(-9));
print("write time[s]: ", 0, 0);
print((t4 - t3)*10.0^(-9));
```

The measurements reported below are the average values of three executions. Finally, in the following figures, we refer to existing kernels for the local filesystem as 'Local FS' to differentiate from our Lustre kernels. Results are depicted in Figure 16 and Figure 17:

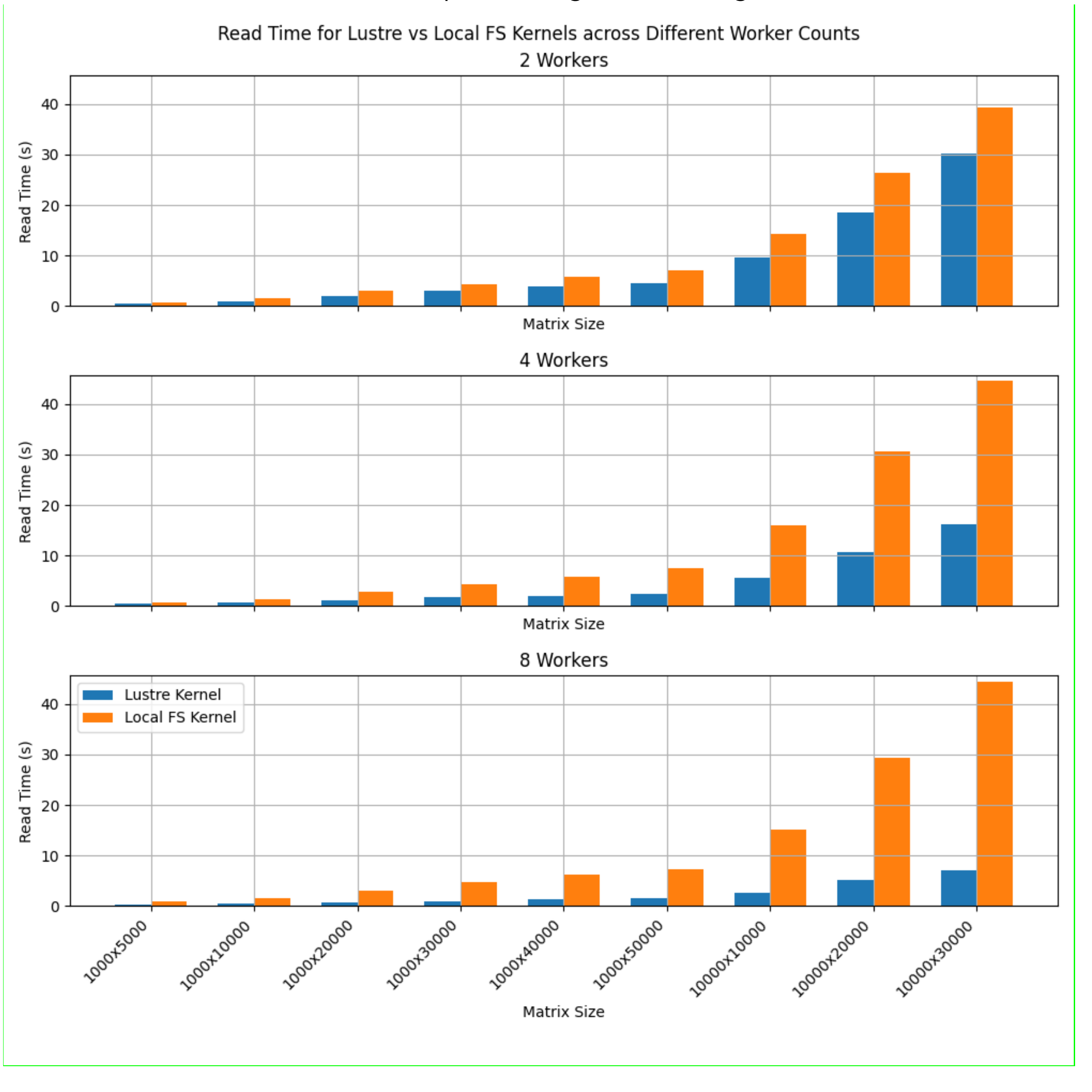

Figure 16: Read time vs. number of worker nodes and matrix size (.csv files)

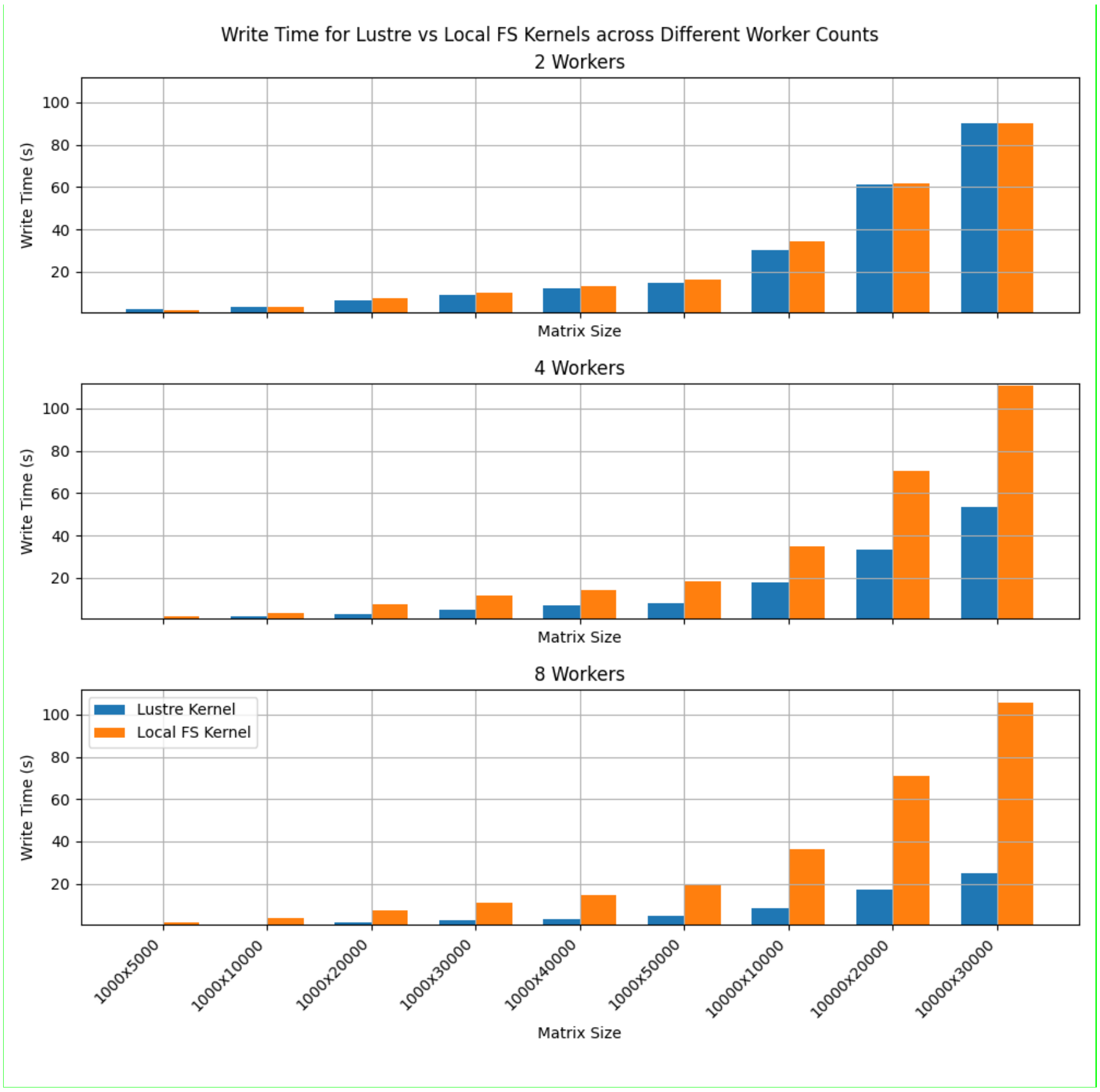

Figure 17: Write time vs. number of worker nodes and matrix size (.csv files)

We observe that, for both read and write operations, the Lustre kernels are at least as fast as the local kernels across all configurations. The difference in performance becomes apparent as the number of workers and the file size increase. This makes sense, as the increased number of workers means better utilization of the parallel attributes of Lustre as well as better utilization of the OSTs.

We also investigate the effect of the number of Lustre stripes on performance by reducing them. This effectively reduces the number of OSTs a file is striped across, making them fewer than the potential workers. For this experiment, each Lustre file is now striped across 4 OSTs compared to 8 OSTs before. Figure 18 and Figure 19 depict the results:

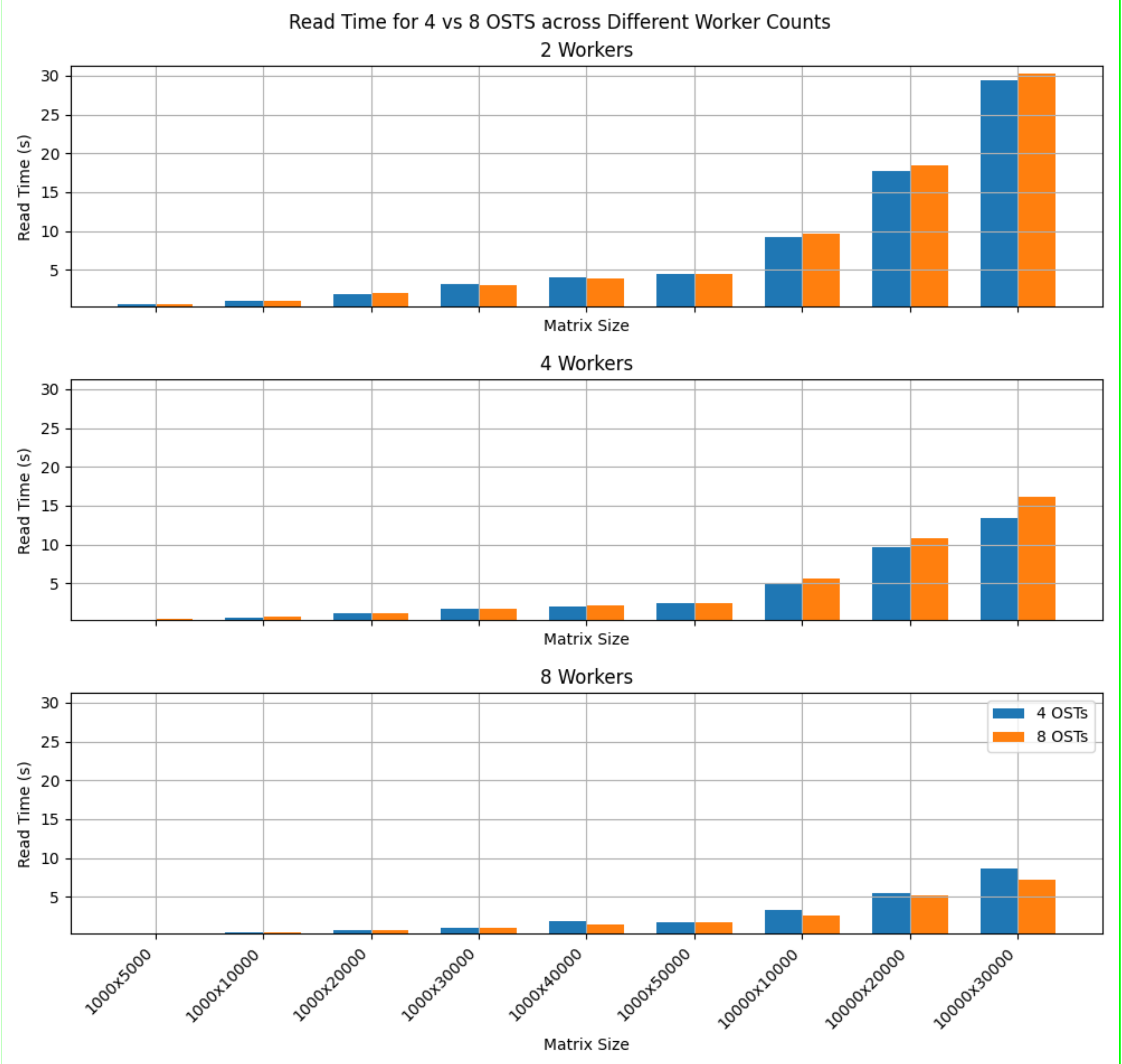

Figure 18: Read time vs. number of worker nodes and matrix size for 4/8 OSTs

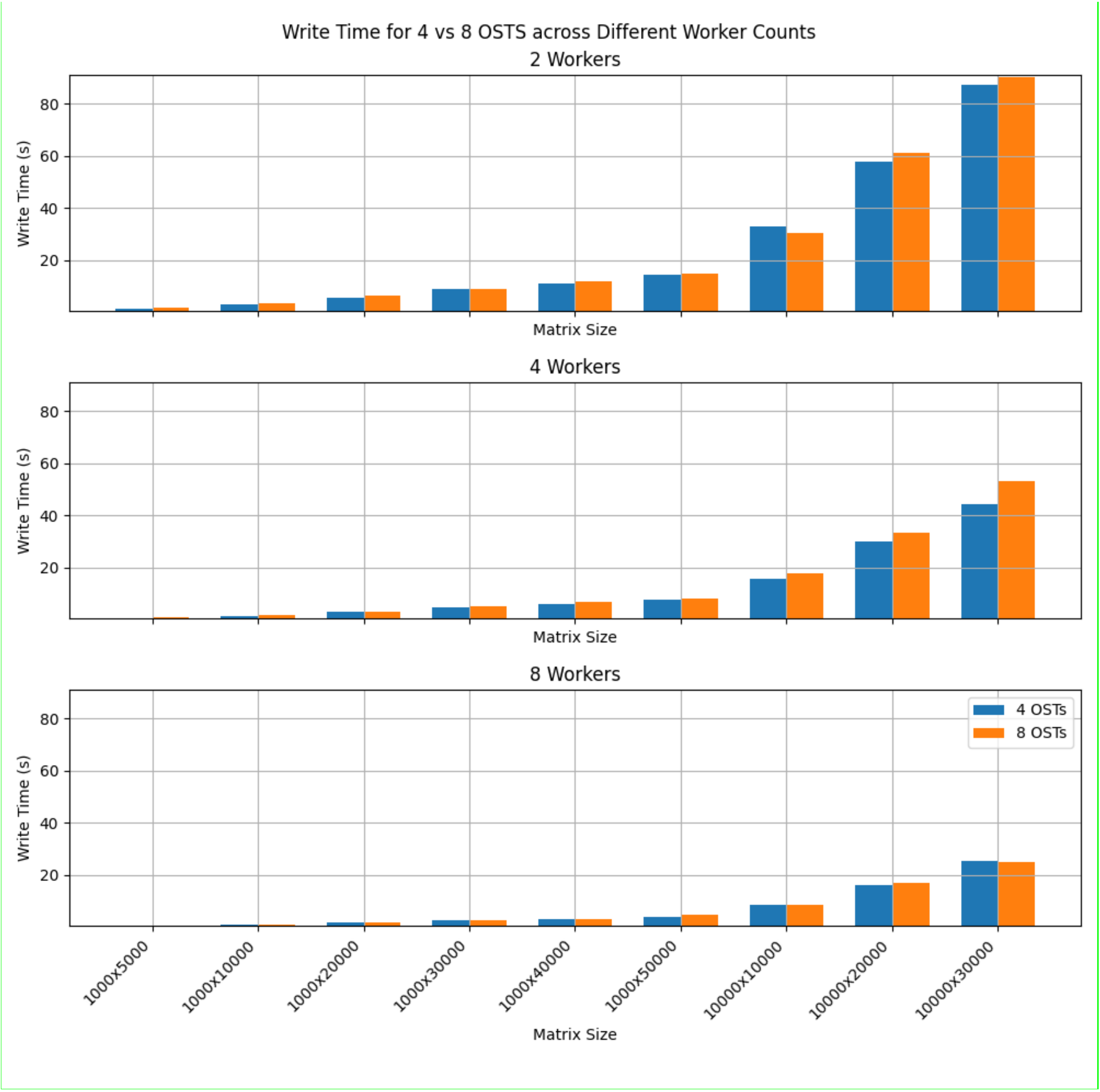

Figure 19: Write time vs. number of worker nodes and matrix size for 4/8 OSTs

As mentioned, the stripe size of 64KB means that all system calls need to access objects from all the OSTs. In that case, the perceived read/write performance is very similar, meaning that increasing the number of OSTs that need to be queried does not affect the overall performance.

To conclude, we observe that apart from the general benefits of a Lustre integration (e.g., widely used in HPC systems), the Lustre kernels are also more performant than local ones. This is true even for naïve/standard configurations currently deployed. In the future, we plan to compare this prototype with the HDFS kernels. Additionally, we would like to investigate whether the system calls can be aligned to require objects from a single OST. This cannot be done without first implementing a more advanced logic for choosing Lustre parameters (mainly the stripe size). This would also enable us to evaluate whether running a DAPHNE worker on top of the Lustre OST can be beneficial by taking into consideration potential data locality.

4 Conclusions

This deliverable demonstrates the progress and enhancements achieved in the DAPHNE DSL Runtime, advancing its capability for large-scale data management, HPC, and machine learning applications. The DAPHNE runtime now integrates key systems like HDFS and Lustre, optimizing data access, distribution, and fault tolerance for both local and distributed setups. Significant improvements in NUMA-aware scheduling and data locality underscore DAPHNE's focus on performance optimization in memory-intensive tasks.

Experiments with varied NUMA policies and task queue layouts revealed critical insights, confirming that memory interleaving combined with localized task distribution enhances performance across multiple nodes, particularly for dense datasets. The evaluation of HDFS integration highlighted the scalability of DAPHNE's distributed I/O, with substantial performance gains as cluster nodes increase.

This prototype delivers a robust foundation for further development and optimization, supporting DAPHNE's goal of delivering a comprehensive, extensible infrastructure for diverse data analysis needs in complex computational environments.

References

[D2.1] DAPHNE: D2.1 Initial System Architecture, EU Project Deliverable, 08/2021.

[D2.2] DAPHNE: D2.2 Refined System Architecture, EU Project Deliverable, 08/2022.

[D4.1] DAPHNE: D4.1 DSL Runtime Design, EU Project Deliverable, 11/2021.

[D4.2] DAPHNE: D4.2 DSL Runtime Prototype, 11/2022.

[D4.3] DAPHNE: D4.3 Improved DSL Runtime Prototype and Overview, 11/2023