D5.4 Final design and Prototype of Scheduling Components

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

Version 1.6

[PUBLIC]

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

Document Description

This report presents the final design and prototype of the scheduling components in the DAPHNE system, as well as performance results. It provides a comprehensive description of the DAPHNE scheduler (local and distributed) for pipelines and tasks, representing the outcome of continuous discussions among all DAPHNE partners, particularly those from WP2 (System Architecture), WP3 (DSL Abstractions and Compilation), WP4 (DSL Runtime and Integration), and WP5 (Scheduling and Resource Sharing). Extensibility remains central to the DAPHNE scheduler, and a robust set of scheduling strategies and techniques have been carefully selected in this final design to support it. The scheduler design allows future scheduling techniques, including user-defined strategies, to be integrated into DAPHNE.

To ensure this report is self-contained, we begin by defining the relevant scheduling terminology and providing an overview of the scheduling-related components within the DAPHNE system architecture, with a focus on the vectorized execution engine. The following sections present the finalized design of the scheduling components and techniques implemented by the DAPHNE compiler and runtime system, along with performance evaluations of the prototype.

As this document presents the final scheduler design, updates to the initial design have been incorporated, marking the completion of the scheduler design and prototype, as outlined in previous deliverables [D5.1], [D5.2], and [D5.3]. To facilitate reading, we include here information that can also be found on previous deliverables.

D5.4 Final design and Prototype of Scheduling Components			
WP5 - Scheduling and Resource Sharing			
Type of document	R	Version	1.6
Dissemination level	PU		
Lead partner	UNIBAS		
Author(s)	Jonas H. Müller Korndörfer, Quentin Guilloteau, Florina M. Ciorba (UNIBAS), Matthias Boehm, Patrick Damme (TUB)		
Contributors	Marius Birkenbach (KAI), Pinar Tözün (ITU)		

Revision History

Version	Comment	Author / Reviewer
V1.0	Outline and initial draft	Jonas H. Müller Korndörfer (UNIBAS) Quentin Guilloteau (UNIBAS)
V1.1	Parameter Server description	Matthias Boehm (TU Berlin) Jonas H. M. Korndörfer (UNIBAS)
V1.2	Prototype and discussion draft	Jonas H. Müller Korndörfer (UNIBAS) Quentin Guilloteau (UNIBAS)
V1.3	Review and editing	Florina Ciorba (UNIBAS)
V1.4	Review	Pinar Tözün (ITU)
V1.5	Review	Marius Birkenbach (KAI)
V1.6	Final review and editing	Jonas H. Müller Korndörfer (UNIBAS) Quentin Guilloteau (UNIBAS) Florina M. Ciorba (UNIBAS)

Executive Summary

This deliverable presents the final design and prototype of the scheduling components within the DAPHNE system, an advanced architecture designed to support integrated data analytics pipelines that combine data management (DM), high-performance computing (HPC), and machine learning (ML). It represents the culmination of collaborative efforts among DAPHNE partners and builds upon prior deliverables by incorporating refinements, enhancements, and a comprehensive evaluation of the developed components.

The report begins by establishing foundational scheduling terminology and describing the role of scheduling in the DAPHNE architecture. By defining concepts like task partitioning, resource assignment, and work timing, it sets the stage for understanding how scheduling decisions optimize performance across shared-memory and distributed-memory systems.

The core of the deliverable focuses on the finalized design of DAPHNE's scheduling framework. Key features include:

1. **User-Centric Scheduling Options**: Users can operate the system with default settings or configure scheduling parameters to suit specific needs. Flexibility is provided through tuning knobs for task partitioning, thread allocation, and resource mapping, ensuring that the system can cater to both non-expert users and advanced use cases.
2. **Compiler-Level Optimizations**: The DAPHNE compiler plays a pivotal role in optimizing performance by reordering operations, fusing pipelines, and deciding on data- and task-level parallelism. These pre-runtime decisions ensure efficient resource usage and adaptability to heterogeneous system architectures.
3. **Runtime Scheduling Mechanisms**: The runtime system incorporates a local scheduler for shared-memory systems and a distributed scheduler for multi-node environments. Advanced features include support for various partitioning strategies, work queue configurations, and adaptive scheduling methods, which balance load and optimize execution.

This report also introduces the **DAPHNE Scheduler Prototype**, which implements these design principles. Detailed usage examples demonstrate the prototype’s flexibility, showcasing its ability to adapt to different workloads and system configurations. For instance, the prototype supports thirteen partitioning techniques, three queue layouts, and four victim selection strategies for task stealing, which users can tailor to their needs.

Comprehensive performance evaluations highlight the benefits of the scheduling strategies. Experiments for both local and distributed runtimes show substantial improvements in execution time and scalability. For distributed environments, the report discusses the design of a coordinator-based architecture that enables efficient data and task distribution across computing nodes.

A dedicated section emphasizes **reproducibility**, outlining methodologies for creating artifacts that align with ACM (Association for Computing Machinery) reproducibility standards. This includes detailed guidelines on environment setup, artifact evaluation, and ensuring repeatability for future research and testing.

The deliverable concludes with a discussion of the system's limitations and potential future enhancements. Suggestions include extending the distributed runtime with additional coordination models, implementing inter-node work stealing, and exploring hybrid scheduling strategies.

This deliverable marks the culmination of scheduling component development within the DAPHNE project, showcasing innovative approaches and providing a robust foundation for ongoing research and application in large-scale data analytics environments.

Table of Contents

List of Abbreviations

Abbreviation	Meaning
AF	Adaptive Factoring
AWF	Adaptive Weighted Factoring
DAG	Direct Acyclic Graph
DG	Directed Graph
DLS	Dynamic Loop Self-Scheduling
DM	Data Management
DSL	Domain Specific Language
FAC	Factoring Self-Scheduling
FSC	Fixed-Size Chunking
GSS	Guided Self-Scheduling
HPC	High Performance Computing
IR	Intermediate Representation
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation
NUMA	Non-Uniform Memory Access
PLS	Performance Loop-based Self-Scheduling
PSS	Probabilistic Self-Scheduling
SIMD	Single Instruction Multiple Data
STATIC	Static Scheduling
SWR	Static Workload Ratio
SPMD	Single Program Multiple Data

1 Introduction

The DAPHNE system architecture—comprising DaphneDSL, the compiler, and runtime—is designed to enable the efficient execution of integrated data analytics pipelines and workflows, including data management and query processing (DM), high performance computing (HPC), and machine learning (ML) training and scoring codes. These workflows are typically executed on distributed- and shared-memory systems with heterogeneous resources (Figure 1), ranging from traditional HPC clusters with shared-disk configurations to shared-nothing systems. Scheduling is a critical component in achieving various performance objectives, such as minimizing execution time, maximizing resource utilization, and increasing computational throughput. The DAPHNE system is no exception; scheduling plays a fundamental role in its overall performance.

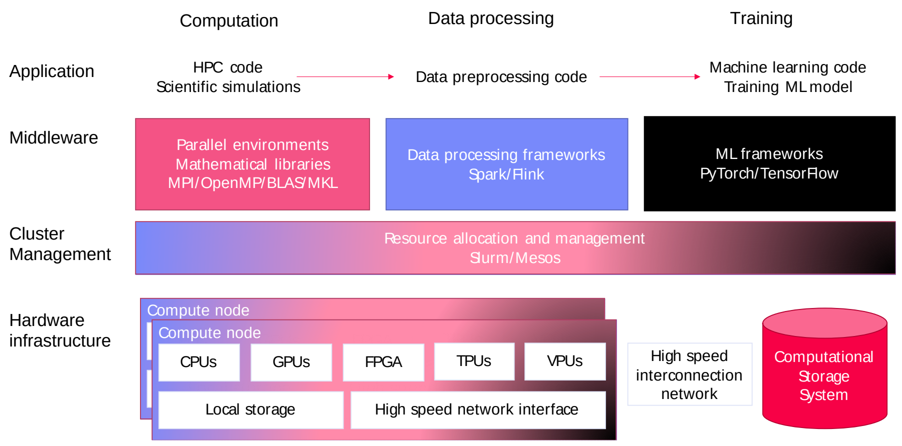


Figure 1 Ecosystem for an integrated data analytics pipeline [IPE+21]

Scheduling involves the process of mapping units of work to computing resources over a defined period of time [BW91] [Ull75]. Solutions to scheduling can be classified into several categories, such as online, offline, optimal, and heuristic approaches. Scheduling encompasses decisions on work partitioning, work assignment, and execution ordering [BW91], with these decisions being made at different levels of hardware parallelism in HPC systems (core, node, and system levels). Depending on when scheduling decisions are made, techniques can range from static to dynamic.

Table 1 Scheduling decisions and implementation details in the DAPHNE system architecture

DAPHNE		**Scheduling Levels**				
		User		**Compiler**	**Runtime System**	
		DSL	**Configuration**		**Local**	**Distributed**
Scheduling Decisions	**Partitioning**	(✓)	(✓)	✓	✓	✓
	Assignment	(✓)	(✓)	(✓)	✓	✓
	Ordering	(✓)	(✓)	✓	(✓)	(✓)
	Timing	N/A	N/A	N/A	✓	✓
Implementation Decisions	**Work Queue**	N/A	N/A	N/A	Centralized	Centralized
					Distributed	Distributed
	Data Placement	N/A	N/A	Centralized	Centralized	Centralized
						Distributed
				Distributed	Distributed	Replicated
	Work Transfer	N/A	N/A	N/A	Receiver/sender-initiated	Receiver/sender-initiated
Optimization Goals	**Minimize**	User-defined	User-defined	Intermediates	Execution time	Execution time
					Scheduling overhead	Scheduling overhead
	Maximize			Data locality	Data locality	Data locality

Work = Task = Operator + Data

Legend: ✓ currently supported | (✓) could be supported in the future | N/A not applicable

Table 1 outlines the various scheduling decisions made by the DAPHNE user, the DAPHNE compiler, and the runtime system. We distinguish between local and distributed runtime system scheduling by emphasizing specific implementation details. This table serves as a guide for describing the techniques employed by the compiler and runtime system, the associated scheduling decisions, and their optimization objectives.

Although the terms in Table 1 are common to different scheduling contexts, one can nevertheless use these terms quite differently. Therefore, we define these terms in the context of DAPHNE scheduling as follows:

Work refers to operations applied to input data.

Work ordering refers to the order in which the operations must be executed, i. e., if the execution of certain operators has data or control dependencies, work ordering must maintain and respect the dependencies.

Work partitioning refers to partitioning of the work into units of work (or tasks) of a certain granularity (fine or coarse) and of certain size (equal or variable). Work partitioning may also exploit **data** and/or **functional parallelism**, i. e., work is partitioned by dividing the input data and then, execution units apply the same operator to each data partition. Work can also be partitioned by enabling each execution unit to execute different operators on the input data.

Work assignment refers to mapping (or placement) of the units of work (or tasks) onto individual software processing units (processes or threads). Work assignment also applies beyond the software level, in the form of mapping specific software units of processing (processes, threads) onto hardware units of execution (compute nodes, CPUs, GPUs, FPGAs) and computational storage devices.

Work timing refers to the times at which the units of work are set to begin execution on the assigned units of execution.

Work queue is an implementation detail that describes how the units of work are managed by the runtime system. Work queues can be centralized or distributed.

Work transfer describes the party initiating the transfer of work, during execution, to arrive at a balanced execution progress. Work transfer can be initiated by underloaded parties (receivers) or overloaded parties (senders).

Optimization goals may refer to minimization of user-defined goals (for the user), number and size of intermediate data items (by the compiler), execution time, and scheduling overhead (by the runtime system). They may also refer to maximization of other user-defined goals (by the user) or data locality (by the runtime system).

1.1 Scheduling Terminology

Scheduling is a *vast* topic [Leu04] and has been an important research focus in the DB, HPC, and ML communities over several decades. As the DAPHNE project has a diverse consortium and specific terms are used differently by the communities, defining a common terminology is extremely important and useful. In the following subsections, we define important terms related to scheduling that we use in DAPHNE.

1.1.1 Operators

The term **operator** refers to an indivisible operation that can be applied to input data. Common examples are matrix operations, such as addition, subtraction, multiplication, and transpose. Calls to user-defined functions or precompiled third-party libraries are also considered operators. We also use the term **kernel** to refer to the actual C++ implementation of a specific operator, e. g., EwBinaryMat or a shallow wrapper around a third-party implementation, e. g., cublasDgemm for Nvidia GPUs.

A **vectorized operator** refers to an operator that can be executed in vectorized form (SIMD), similar to Single Program Multiple Data (SPMD).

1.1.2 Pipelines

The term **pipeline** refers to an abstraction that combines one or more operators (i.e., fused operators). A **vectorized pipeline** consists of one or multiple vectorized operators which can be applied to vectorized data.

1.1.3 Tasks

A **task** comprises a data item and an operator to be applied. Tasks are the smallest units of work considered for scheduling by the DAPHNE runtime system.

A **vectorized task** consists of a data partition and a vectorized pipeline.

1.1.4 Workflows

The term **workflow** is commonly used in different communities (DM, HPC, and ML) to refer to execution of work in a specific order. The most common approach of representing workflows is to use graphs where each vertex represents a task (work step) and an edge represents a data or a control dependency [VA20]. Workflows can be cyclic (represented as Directed Graphs - DG) or **acyclic** (represented as Directed Acyclic Graphs - DAG), hierarchical (a workflow within a workflow, represented by a **hierarchical** D[A]G), and may form **workflow ensembles** (sets of interrelated workflows also expressed as a D[A]G) [D+19]. Since the DAPHNE compiler is based on MLIR, we represent workflows as DAGs. Figure 2 shows an example of a workflow highlighting relevant terminology for different workflow levels (adapted from [D5.1] and [D8.1]).

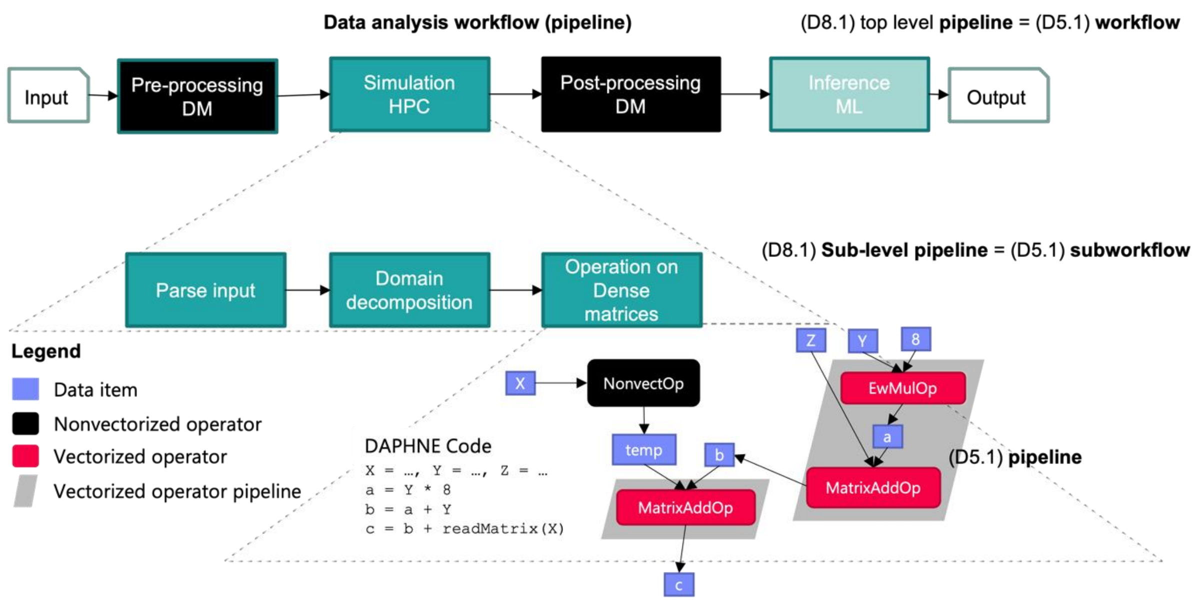


Figure 2 Data Analysis Workflow and the Associated Terminology at Various Levels. Figure adapted from D5.1 and D8.1.

1.2 Scheduling Classes

Scheduling can be categorized into two main classes: **static** and **dynamic** scheduling. The main difference is when decisions are taken. In **static scheduling** decisions are taken before execution (at compilation time), while in **dynamic scheduling** decisions are taken during the execution of an application. Between these two main classes, hybrid scheduling exists. A hybrid scheduling technique is not fully-static nor fully dynamic, i. e., certain decisions are taken before the execution, while others are taken during the execution.

Static scheduling techniques incur minimal scheduling overhead and may be explicitly designed to improve data locality [LTS+93]. **Dynamic scheduling techniques** are explicitly designed to improve load imbalance [Luc92] but also to allow scheduling of work under dynamically evolving conditions in the application, the system, or both.

Dynamic scheduling techniques can further be divided into **nonadaptive** and **adaptive**. **Nonadaptive dynamic scheduling** techniques take scheduling decisions (such as work partitioning, assignment, and timing) during execution but do not change these decisions once taken. For instance, a nonadaptive scheduling technique may require runtime information about the underlying system, e. g., processors' speed. Such information can be obtained prior to the execution and used to calculate the work amount each processor receives. In contrast, **adaptive dynamic scheduling** techniques consider information obtained during execution to refine their scheduling decisions, i. e., processors' speed may change during the execution (e.g., DVFS, Linux governors). The adaptive dynamic scheduling techniques are explicitly optimized to minimize load imbalance in highly irregular execution environments [Ban00]. Nonadaptive dynamic scheduling incurs less overhead during execution than adaptive dynamic scheduling.

1.3 Scheduling Levels

A DAPHNE program is a workflow, i. e., a DAG of operators (which may originate in DM, HPC, and/or ML codes), their associated data, and a specific data flow. Allocating resources and executing such a DAG on those resources is subject to various scheduling decisions, that are taken at different levels: user, compiler, and runtime system.

1.4 Summary

This section introduces the DAPHNE system architecture, which supports efficient execution of data analysis pipelines, including data management, high performance computing, and machine learning (ML). The section highlights the critical role of scheduling in achieving key performance goals such as reducing execution time, enhancing resource utilization, and improving throughput. It also carefully defines important scheduling terminology, including concepts such as work partitioning, assignment, and timing. These definitions and distinctions lay the groundwork for the later sections, which build on this common understanding.

2 Final Design of Scheduling Components and Extendibility

2.1 Scheduling by the User

By default, all scheduling-related decisions will be made by the DAPHNE system automatically (based on defaults provided by the DAPHNE developers), as described in Sections 2.2 and 2.3. However, expert or interested users may optionally configure the scheduling behavior of some DaphneDSL program manually through command line arguments. Available tuning knobs include the number of local threads and distributed workers, the task partitioning technique and its parameters, the placement of data and operations on certain (classes of) devices, and generally the modification of compiler's behavior (including the sequence of compiler passes by turning certain passes on or off) through compiler flags. For details on the different available parameters that configure the DAPHNE local and distributed scheduler refer to Section 3 which describes the final prototype and provides usage examples.

2.2 Scheduling by the Compiler

The input to the DAPHNE system architecture is typically a DaphneDSL source code (possibly generated from calls to DaphneLib, the Python API for the DAPHNE system), which is a declarative representation of a program and specifies its intended semantics [BE+16]. A parser translates this into DaphneIR as the central representation for reasoning about a program at compile-time. The initial DaphneIR representation of the user program will usually not yield an efficient runtime behavior. Thus, the DAPHNE compiler exploits the declarative nature of the user program to perform various rewrites on the intermediate representation (IR) of the program, which preserve its semantics but allow for a more efficient execution.

In particular, the DAPHNE compiler determines:

(a) which operations to execute,

(b) in which order to execute them (work ordering) as well as

(c) selective aspects of whether and how to parallelize the work (work partitioning) and which classes of devices (e. g., CPU, GPU, FPGA) to assign tasks to (work assignment). Details on the

DAPHNE compiler can be found in D3.4 that includes the compiler design and overview; here we only provide a brief overview and highlight how compiler decisions relate to scheduling.

2.2.1 Reordering Rewrites

The DAPHNE compiler applies various **static** and **dynamic simplification rewrites** on the IR, which include the removal, insertion, exchange, and reordering of operations under different goals. This can lead to a reordering of the entire DAPHNE program.

Some rewrites aim at **eliminating redundant operations**, or **reducing the number of operations**. These goals are addressed by **generic compiler optimization techniques**. For instance, common subexpression elimination (CSE) eliminates redundant calculations of the same expression (if there are no side-effects), and constant propagation performs simple calculations that can be evaluated at compile-time to gain more information on concrete inputs to operations. This may enable the elimination of branches, thereby significantly changing the program structure.

Complementarily, **domain-specific simplifications** from linear algebra [BBE+14] and relational algebra [EN16] are applied with goals such as minimizing the memory footprint and the execution time. In that context, **reducing the size of intermediate results** is a natural objective, and for many operations, smaller inputs incur a lower effort. Examples from linear algebra include the optimization of chains of arithmetic operations over scalars, vectors, and matrices as well as matrix multiplication chain optimization [HS82]. The latter exploits the associativity of matrix multiplication to freely choose the parenthesization to reduce the size of intermediates. This rewrite is based on complex algorithms, but has the potential of speeding up the calculation by orders of magnitude. Examples from relational algebra include selection push-down, whereby predicates on a single relation can be evaluated before a join with another relation to reduce the size of the join inputs. Furthermore, join ordering also exploits associativity. Different join orders can result in intermediate size and runtime differences by orders of magnitude, which justifies the employment of sophisticated optimization techniques [HHH+21] [LGM+15] [LRG+17].

While most rewrites view a DaphneIR program as a DAG of operations connected through their inputs and outputs, this DAG must be linearized (work ordering) to be executable by each single worker out of multiple workers. Thus, another aim is to reorder operations by defining an **efficient linearization** of the program. Any topologically sorted order would be valid, but the DAPHNE compiler will aim at finding a linearization that increases temporal and spatial **locality of data accesses**, e. g., by ordering operations reading the same data close to each other. That way, the cache behavior may be improved and evictions from the buffer pool to secondary storage avoided.

Finally, **the selection of physical operators** has a strong impact on the execution time and the working memory footprint of an operation. Furthermore, the choice of the physical operator can have an impact on the applicability of data partitioning and multi-threading execution techniques, which are the basis for scheduling during the execution time.

2.2.2 Pipelines and Operator Fusion

In the simplest case, the reordered sequence of operations could be executed in an operator-at-a-time fashion, i. e., operations are executed one by one and each operator materializes its entire output data objects in the storage hierarchy. This approach is adopted by machine learning frameworks such as TensorFlow [ABC+16] and database systems such as MonetDB [IGN+12]. However, to avoid unnecessary materialization overhead, the DAPHNE compiler tries to fuse adjacent operations together into pipelines whenever possible. From outside, a **fused pipeline** looks like a single operation with a number of inputs and outputs that depends on the operations within the pipeline. During execution, a pipeline's inputs are split into tiles (partitions). The DAG of operations within the pipeline is executed for the set of corresponding tiles of each input, i. e., as a **vectorized execution** [BZN05]. This partitioning ensures a cache-efficient behavior if the partition sizes are chosen to fit the intermediates within the pipeline into the cache hierarchy. In combination with a multi-threaded execution, this leads to an execution akin to morsels [VL14].

Note that the compiler already decides whether and how (along which dimension(s) such as row/column/both, as of now only a partitioning per row is possible) to **partition the data**. For instance, for the elementwise addition of a matrix and a row vector, one option would be to partition the matrix horizontally, while broadcasting the row vector. While the compiler decides the task partitioning technique (see Section 2.3), the actual partitioning is performed at run-time. In general, the fused operator pipelines created by the DAPHNE compiler define the unit of work for the DAPHNE runtime scheduler.

2.2.3 Decisions about Data-level Parallelism

The DAPHNE compiler performs intra- and inter-procedural analyses [BBE+14] to compute the dimensions, and estimate sparsity, and other properties of intermediate results. Based on these, appropriate physical data representations (such as dense or sparse matrices) are selected and memory footprints estimated. Depending on the physical size of the intermediates, the compiler decides **if parallelization is required**, whereby different levels are supported. For very small data objects, the overhead of parallelization might outweigh its benefits, e. g., due to thread setup or data transfer costs, rendering a sequential processing most efficient. However, the DAPHNE system is specifically designed for processing large amounts of data. Within a single computing node, multi-threading is applied to process different tasks of a vectorized pipeline in parallel using the **local runtime**. If the expected size of an intermediate result exceeds the memory capacity of a single computing node, the pipeline will be executed on several computing nodes by the **distributed runtime**. That means, the compiler decides if and at which level to parallelize a pipeline, thereby triggering either the **local** or the **distributed runtime scheduler**. This behavior is also controlled and influenced by the user as the distributed runtime must be activated to be used (see prototype Section 3).

2.2.4 Decisions about Task-level Parallelism

Apart from data-level parallelism, machine learning algorithms, as a core component of integrated data analysis pipelines, often expose task-level parallelism in the form of a *for loop* with independent loop iterations. This can be the case when training multiple models (e. g., in ensemble learning) as well as when training a single composable model (e. g., in stochastic gradient descent). The individual loop iterations could access disjoint or overlapping parts of the data, or the entire data. Thus, when executing subsets of the for loop's index range in a task-parallel manner, different techniques like data partitioning or memoization/sharing could be applied. Similar to SystemML [BTR+14], the DAPHNE system will support ParFOR loops (described in Section 2.1) to explicitly express opportunities for task-level parallelism. The DAPHNE compiler plays a crucial role by ensuring that there are no dependencies between iterations and by selecting an optimal task-parallel execution strategy for minimizing the overall runtime under hard constraints on the memory consumption and the degree of parallelism. In that sense, local, distributed (remote), and hybrid (local and distributed) execution is possible, depending on the data size.

2.2.5 Code Generation

By default, the operations within a fused pipeline are executed by the same runtime kernels used for a stand-alone operator outside a fused pipeline, whereby efficiency is achieved through cache-awareness and multi-threading. In the literature, approaches for the on-the-fly generation and JIT-compilation of **tailored operators** have been investigated for ML systems, such as SystemML [BRH+18] and Julia [BEK+17] and database systems, such as HyPer [N11]. The DAPHNE compiler adopts these approaches to specialize operators for the actual data and value types, shapes, and sparsity of the data, as well as for the hardware to use. However, since code generation incurs a certain extra effort during execution time, it will be applied in a cost-beneficial manner [BRH+18]. By creating tailored operators on-the-fly, the compiler further defines the scope of the runtime scheduler.

2.2.6 Placement

By means of configuration (for more details, see [D3.1] and [D3.4]), the DAPHNE compiler is aware of the (heterogeneous) accelerator devices available to the system. It may automatically determine on which class of devices an entire operation should be placed. Moreover, the DAPHNE system will support the execution of vectorized pipelines by **heterogeneous worker nodes**; while the fine-grained assignment of tasks to threads and of threads to workers are subject to the runtime scheduler, the compiler makes important preparations by optimizing and JIT-compiling separate copies of the pipeline's body for each hardware device it might be assigned to during the execution time.

2.3 Scheduling in the Runtime System

The DAPHNE runtime system manages work and data scheduling through two main components. The first component is the local runtime with the vectorized execution engine, responsible for creating tasks and assigning them to C++ worker threads (see Section 2.3.1). The second component is the distributed runtime (see Section 2.3.2), which, when enabled (refer to Section 3 for usage details), handles the distribution of work and data across multiple MPI or gRPC processes.

2.3.1 Scheduling in the Local Runtime

The local DAPHNE scheduler relies on a vectorized execution engine which was initially described in [D5.1] and [D4.1]. Figure 3 shows the complete workflow of the local DAPHNE runtime scheduler.

(1) The work partitioner merges operators and data to compose tasks of different sizes.

(2) The work manager iteratively queries the work partitioner for more tasks until all data is consumed. The work manager also receives information from the compiler about the type of available workers and tasks. Finally, it initializes the workers and queues the tasks into different queue types depending on the worker type and queue type configured by the user (or default centralized queue).

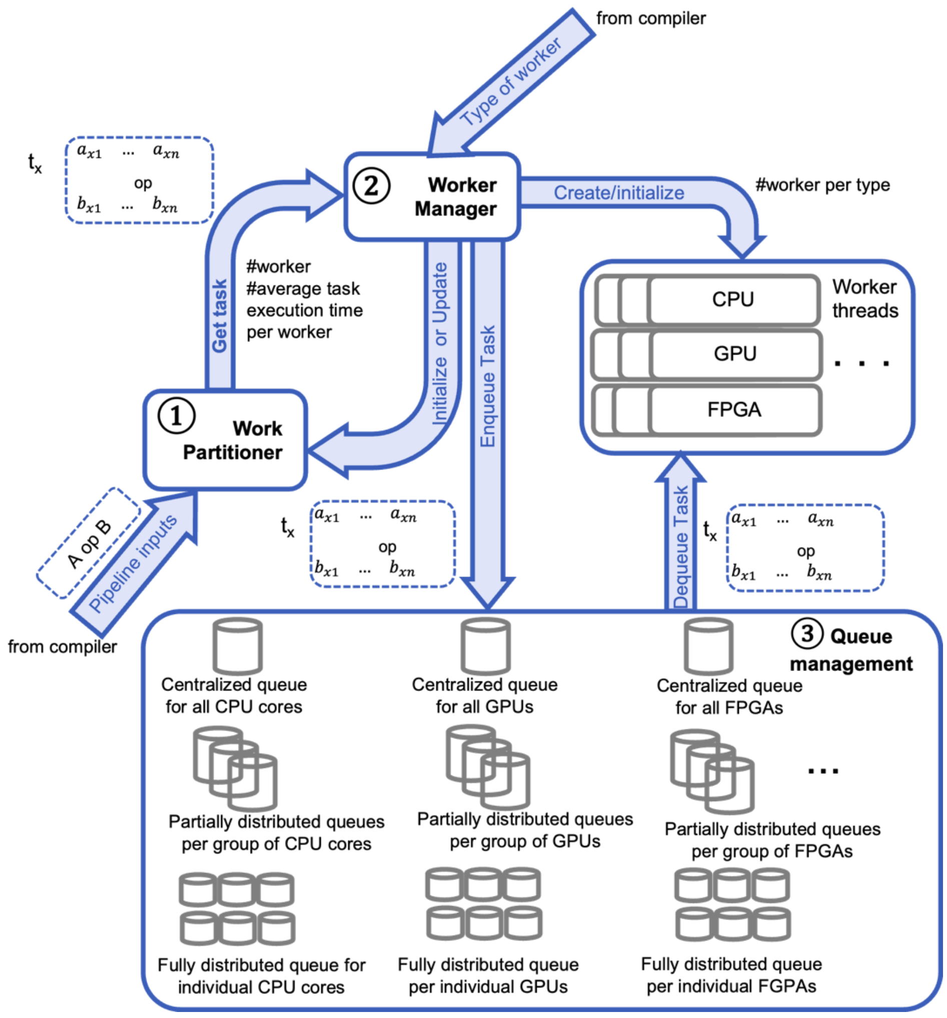

from compiler
Type of worker
② Worker Manager
Create/initialize
#worker per type
t_x a_{x1} ... a_{xn} op b_{x1} ... b_{xn}
Get task
#worker
#average task
execution time
per worker
Initialize or Update
Enqueue Task
CPU
GPU
FPGA
Worker threads
① Work Partitioner
A op B
Pipeline inputs
from compiler
Dequeue Task
③ Queue management
Centralized queue for all CPU cores
Centralized queue for all GPUs
Centralized queue for all FPGAs
Partially distributed queues per group of CPU cores
Partially distributed queues per group of GPUs
Partially distributed queues per group of FPGAs
Fully distributed queue for individual CPU cores
Fully distributed queue per individual GPUs
Fully distributed queue per individual FGPAs

Figure 3 Local DAPHNE runtime scheduler workflow for work and data partitioning, task creation, task queuing, and task assignment. Figure adapted from [EC23].

The DAPHNE vectorized execution engine simplifies scheduling complexity during execution, i. e., all software units of processing (threads, processes) and hardware units of execution (CPUs, GPUs, FPGAs), execute the same operators on multiple data chunks (SPMD). This strategy brings certain simplifications to the runtime system, i.e., no data dependencies among vectorized tasks similar to DOALL loops (parallel loops with no loop-carried dependencies, allowing iterations to execute independently). DAPHNE leverages data parallelism, where the input data is divided into partitions, and the same operation (or a small set of operations) is applied concurrently to each partition, as shown in Figure 4.

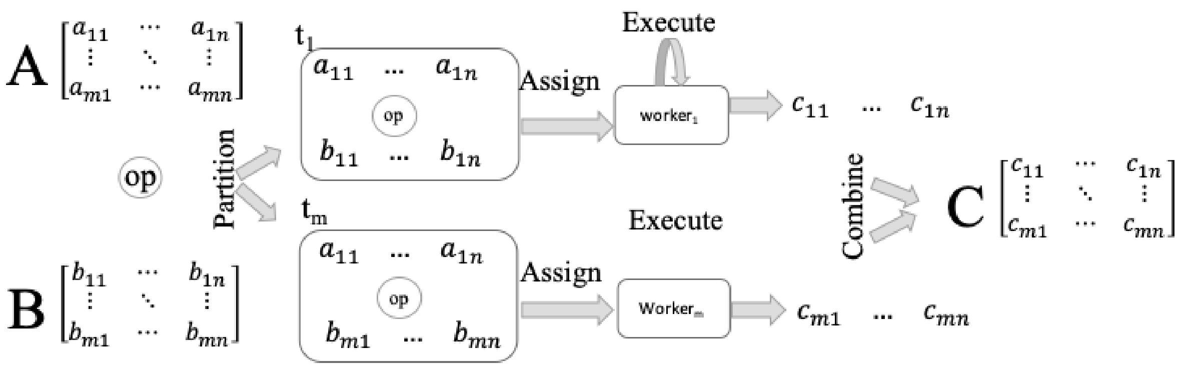


Figure 4 Data parallelism in the DAPHNE runtime system. This figure is taken from [EC23]

Applying data parallelism to sparse data poses challenges because the execution time of an operator depends on (1) the size of the data partition, (2) the hardware executing the operator, and (3) the execution order, which must consider spatial data locality. The DAPHNE local scheduler tackles these challenges in the following way.

2.3.2 From Data to Tasks

DAPHNE is a **task-based system**, where its scheduler converts inputs from the vectorized execution engine, namely data and operators, into tasks. Tasks are the smallest unit of work to be scheduled. The work granularity or task granularity is dictated by the size of the data. Since DAPHNE relies on dense and sparse matrix data structures [PD22], the smallest data size can be one row, one column, or a chunk of rows and columns of a certain size. In fact, the proper data size should be considered based on the size of the lower levels of cache of the target system. For simplicity, one can assume that the smallest data within a task is one row. The strategy of creating and executing fine-grained tasks minimizes load imbalance, but it incurs a high overhead that can increase the execution time.

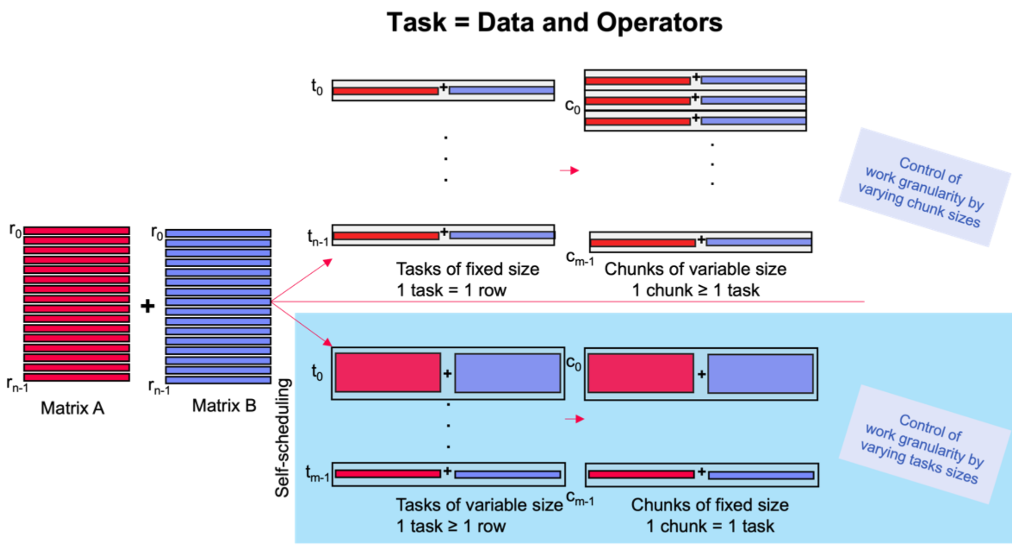


Figure 5 Data partitioning into tasks in the DAPHNE local runtime scheduler. Figure adapted from [D5.1]

One way to address this challenge is to still create fine-grained tasks but to schedule them in chunks of tasks (see Figure 5). This approach reduces the overhead of scheduling individual tasks. Another approach is to create tasks of variable sizes (see Figure 5). The local DAPHNE scheduler uses the latter approach (highlighted with blue background in Figure 5) to avoid an unnecessary level of abstraction (chunks of tasks). To determine the task size, the local DAPHNE scheduler employs **self-scheduling techniques** (see next section) to partition the work and data.

2.3.3 Task Partitioning

The DAPHNE local scheduler task partitioner has two interface points: 1) *Initialize/Update* which sets the number of workers (threads), the partitioning scheme, and the total number of tasks. This point is also used to give updates regarding the runtime information to the partitioner. 2) *Get Task* which provides a task for execution. The partitioner is used iteratively (see Figure 3) to support the extension of the DAPHNE local scheduler with dynamic and adaptive scheduling techniques. Task partitioning can thus be adaptive, based on runtime information or non-adaptive following a given scheduling technique chunk size progression. The final local DAPHNE scheduler prototype supports twelve non-adaptive dynamic scheduling techniques and one automated chunking method (also non-adaptive) that can be chosen by the user to govern the partitioning of data into tasks. The available scheduling techniques are STATIC (default), SS, MFSC, GSS, TFSS, FAC2, TFSS, FISS, VISS, PLS, PSS, MSTATIC, and the automated chunking method is named AUTO. The detailed descriptions of these techniques and algorithms can be found in [D5.1] (all techniques) and [D5.3] (AUTO).

2.3.4 Worker Management

The local DAPHNE scheduler is designed to support different types of computing resources, e.g., CPUs, GPUs, FPGAs, and computational storage. The decision of which type of computing resource will be used to execute a specific pipeline is taken by the compiler. The worker manager initiates workers (threads) that execute or interface with the corresponding devices (see Figure 4, worker manager). For example, for CPU workers, the worker manager creates/manages worker threads that execute the tasks on CPUs. Also, it creates worker threads that perform data transfers and launch kernels on target devices, such as GPUs and FPGAs.

Queue management. The number of generated tasks is often larger than the number of available workers. Therefore, the tasks need to be stored for later execution by worker threads. The DAPHNE local scheduler offers a queuing system that stores tasks until workers become free to execute them. It also offers multiple queuing strategies, as shown in Figure 6.

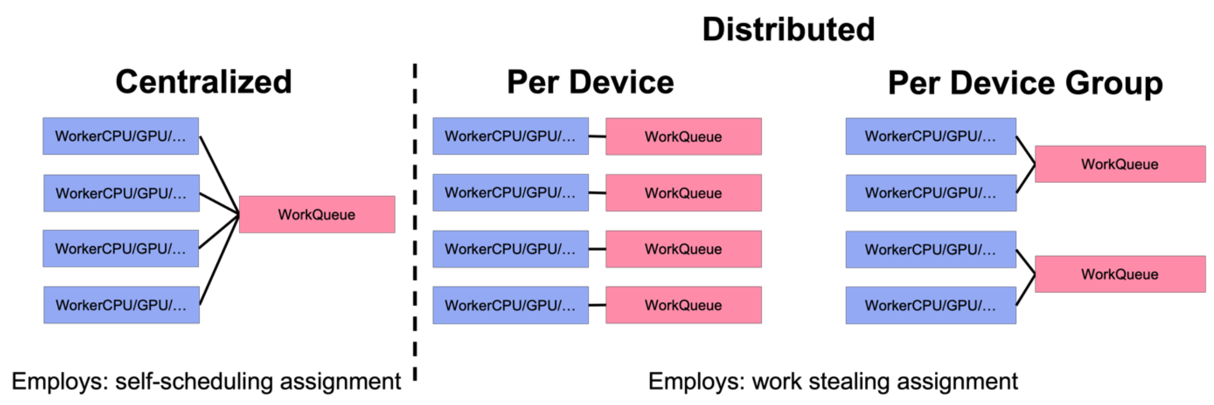


Figure 6 Task queueing strategies in the local DAPHNE scheduler.

The **"Centralized"** queueing strategy relies on creating a single queue per type of computing resource. When the DAPHNE compiler decides to map a particular operation to a specific device (CPU, GPU, FPGA), it generates the corresponding code that executes on the chosen device type. If the centralized queue strategy is being used, all tasks associated to a given type of device will be queued in a single queue.

The **"Per Device"** queueing strategy relies on creating a queue for each device, e.g., in the case of CPUs, each CPU will have a respective queue where tasks are stored. That is, in general, each worker has a separate (local) work queue from which it self-obtains tasks. Once its work queue is empty, the worker seeks tasks from other workers (see work assignment Section 2.3.5).

The **"Per Device Group"** queueing strategy relies on creating a queue for each device group, e.g., one queue per NUMA domain. This type of queue is mainly to support NUMA-awareness, i.e., cores that belong to the same NUMA-domain can share the work queue of that domain as the cost of obtaining tasks from this queue is lower compared to obtaining tasks from a queue residing on a different NUMA-domain (see work assignment Section 2.3.5).

All work queues are initially filled statically (see Figure 3). That is, the DAPHNE local scheduler initially partitions (task partitioning) the data into tasks of different sizes following a given scheduling technique and then fills all queues (or a single centralized queue) with such tasks. Finally, the queues are consumed by the threads following different work assignment strategies.

2.3.5 Work Assignment

Work assignment refers to the mapping (or placing) of units of work (or tasks) onto individual units of execution (processes or threads). In the DAPHNE local scheduler, the work assignment is independent of task granularity (defined by task partitioning), i. e., the task granularity can be identified before the actual assignment and execution of the task. For work assignment, we consider the self-scheduling principle [WS81] which means once an execution unit is free and available it obtains a collection (or chunk) of tasks to execute [EC19]. In general, there are two approaches for work assignment: work sharing and work stealing [CG17]. Both approaches follow the self-scheduling principle and are implemented with centralized and/or distributed work queues. Figure 7 shows the different work assignment strategies depending on the type of work queue being used.

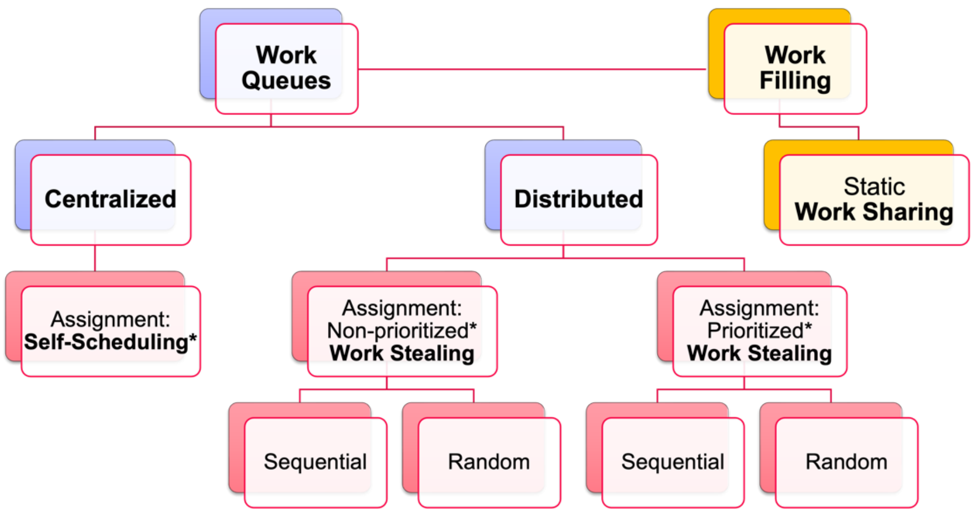

Work Queues
Work Filling
Centralized
Distributed
Static Work Sharing
Assignment: Self-Scheduling*
Assignment: Non-prioritized* Work Stealing
Assignment: Prioritized* Work Stealing
Sequential
Random
Sequential
Random

Figure 7 Task assignment strategies in the local DAPHNE scheduler.

When using a **centralized work queue**, once a worker thread becomes free and available it obtains a new task to execute from the centralized work queue. Such strategy has the following **advantages:** 1) simple design and implementation and 2) a centralized work queue maintains a global overview of the remaining work, and thus, enables load balancing across all workers. The **disadvantages** of consuming work from a single centralized queue are: 1) when the number of workers increases, access to the centralized work queue becomes a synchronization bottleneck that may negatively impact performance and 2) loss of data locality.

For **distributed work queues**, the assignment of tasks to workers works in two separate steps. Initially, while the distributed queues (either per device or per group, Figure 6) are still containing tasks, the threads consume their local queues in the same way as for centralized queue configuration. When the local distributed queues are empty, **work stealing** is applied where threads will steal work from other queues until all tasks are consumed from all queues. The **advantages** of distributed queues with work stealing are: 1) relieves the contention associated with concurrent access to a centralized work queue and 2) is data locality-aware. The **disadvantages** of this approach are: 1) the workers lack global knowledge of the remaining work to execute, and thus, only enables *local load balancing* among specific thief-and-victim workers and 2) requires a more complex design and implementation than work sharing.

Work-stealing and victim selection. In the work stealing approach, when a worker finishes its local work queue it becomes a thief which needs to find a victim to steal work from. Various victim selection strategies are possible. In the local DAPHNE scheduler we provide four options, Sequential victim selection (SEQ), sequential prioritized victim selection (SEQPRI), random victim selection (RND), and random prioritized victim selection (RNDPRI). SEQ denotes a worker searching for the victim in a round-robin fashion, starting from their current position in the system topology [PS14]. SEQPRI prioritizes the search for victims within the same NUMA domain. SEQPRI preserves data locality between NUMA domains whenever possible, and minimizes inter-socket communication [CG17]. RND involves a random selection of one victim using uniform random distribution over all victims. RNDPRI is similar to RND except that victims are divided according to their NUMA domains, i.e., RNDPRI randomly selects from the victims within the same NUMA domain.

2.3.6 Extendibility

The choice of the work partitioning scheme has a strong impact on the performance of applications. For instance, when work stealing is chosen as a work assignment scheme, the granularity of the stolen tasks is determined by the chosen work partitioning scheme. Thus, the choice of the work partitioning scheme indirectly influences the work assignment schemes. **The local DAPHNE scheduler is extendable** as users are allowed to modify any existing or add any new partitioning schemes. DAPHNE developers can use a detailed guide available on GitHub[1] which provides step-by-step instructions on how to modify the runtime for the implementation of additional partitions schemes.

2.3.7 Scheduling in the Distributed Runtime

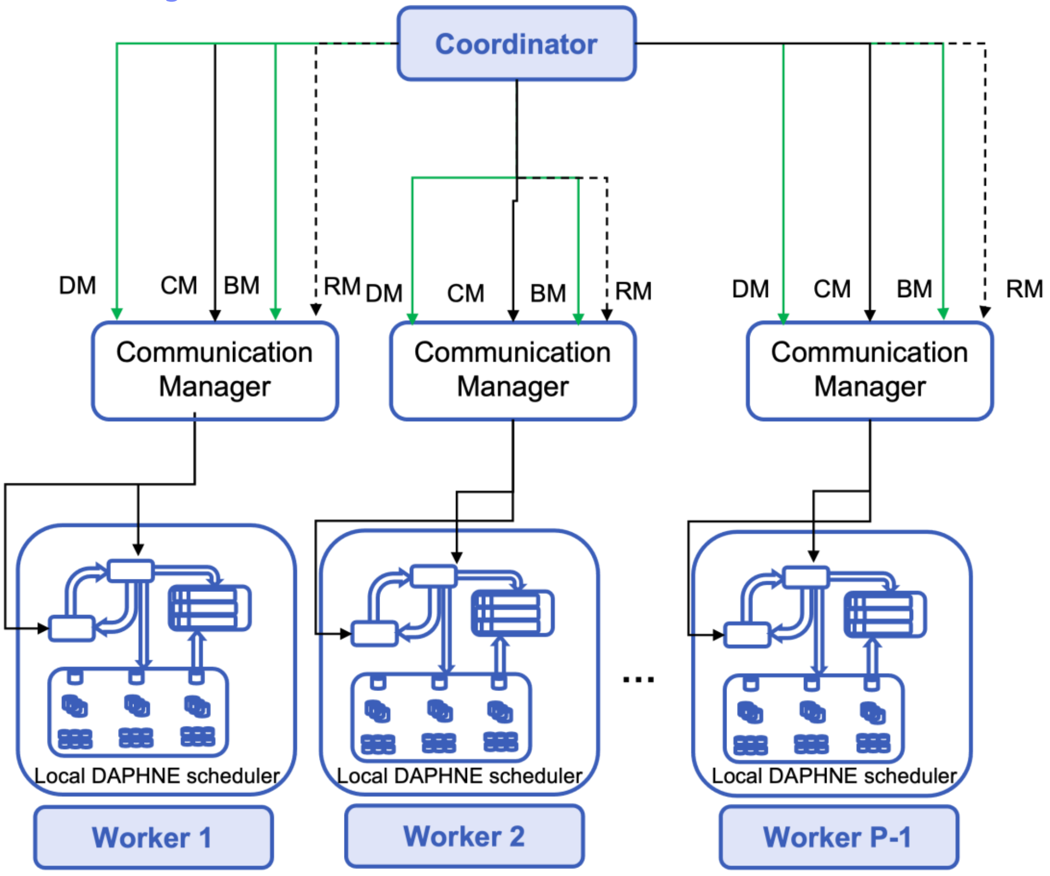


Figure 8 Distributed DAPHNE runtime scheduler workflow for work and data partitioning. Figure adapted from [EC23].

The **distributed DAPHNE scheduler** controls the scheduling of data and work to processes distributed across computing nodes using MPI or gRPC. Figure 8 shows the design extension

[1] https://github.com/daphne-eu/daphne/blob/main/doc/development/ExtendingSchedulingKnobs.md

of the local DAPHNE scheduler to support distributed-memory systems. The design reuses the local DAPHNE scheduler for shared-memory systems. The distributed DAPHNE scheduler adds a new component called coordinator that interfaces with multiple instances — one per MPI process — of shared-memory DAPHNE schedulers. The coordinator serves as an entry point that the DAPHNE runtime uses, i.e., the DAPHNE runtime system replies to the coordinator to divide, distribute, and collect tasks and results from individual local DAPHNE scheduler instances.

The distributed DAPHNE scheduler implements different **message types** interfaced by a communication manager (see Figure 8): Broadcast Message (BM), Compute Message (CM), Distribute Message (DM), and Ready Message (RM). **BM and DM** messages are used to communicate **data** to the distributed processes. The decision of which type of message (BM or DM) will be used to communicate the data is taken by the compiler depending on whether such data is required in every process or if it can be partitioned separately. **CM** messages are used to communicate **MLIR** code to the distributed processes. Finally, **RM** messages are used for **synchronization** between the coordinator and workers.

Distributed partitioning. The current implementation of the DAPHNE distributed scheduler partitions the input data in the coordinator using *equally sized chunks* (STATIC). Then, the data and MLIR code associated with it are communicated — one chunk per worker — to the workers' processes. Finally, each worker process schedules its threads using its own instance of the local DAPHNE scheduler.

2.3.8 Extendibility

The distributed DAPHNE scheduler offers **ample extensibility potential**.

1) The scheduling coordination can be modified to support multiple coordinators or fully distributed coordination.

2) To ensure load balancing beyond the local DAPHNE scheduler, work stealing can be implemented across the distributed processes. The current implementation of work stealing for the local DAPHNE scheduler can be extended for the distributed scheduler.

3) Additional partitioning strategies can be implemented in the distributed DAPHNE scheduler. Guidelines on extending the distributed DAPHNE scheduler can be found on the DAPHNE GitHub repository[2].

2.4 Parameter Server

In this section, we first summarize the background of data- and model-parallel parameter servers, typical synchronization strategies and techniques for reducing the communication overhead. Subsequently, we then summarize our work on the BAGUA prototype and various parallelization strategies in new environments. Since Ce Zhang has left ETH Zurich in the middle of the DAPHNE project, these updated prototypes were explored standalone and are not integrated into the DAPHNE system infrastructure yet.

[2] https://github.com/daphne-eu/daphne/blob/e93c94345d3e6a79ff3bebde34c577c87a43278e/doc/development/ExtendingDistributedRuntime.md

Background: Data- and model-parallel parameter servers, and similar distribution strategies, are the predominant system architecture for mini-batch training of deep neural networks (DNN) and other expensive ML models such as Large Language Models (LLMs) and foundation models [YH+22] in general. In a classical data-parallel parameter server [S+10, D+12, Li+14], we have M parameter servers (collectively holding the model weights), and N workers (each holding a disjoint partition of the data). Starting from a randomly initialized model, the workers pull the current model from the parameter server(s), perform one or multiple forward and backward passes on data batches to compute gradients, and push these gradients or local models back to the parameter server(s) where the gradients are aggregated and models are updated accordingly. Similar architectures apply to multi-threaded training [ZR14], multi-device training [TF19], distributed training [D+12], as well as federated training [KMS20].

Synchronization Strategies: Commonly applied update strategies for when to update the models include Bulk Synchronous Parallel (BSP), Asynchronous Parallel (ASP), and Synchronous with backup workers [D+12, A+16, J+17]. BSP introduces barriers and waits for updates from all workers which ensures consistency but is prone to stragglers (slow workers) because workers wait for the slowest. In contrast, ASP avoids barriers by immediately updating and returning the model for every worker update. However, in case of slow workers, there is the problem of working on stale models which can slow down the learning process, and in extreme cases, even lead to divergence. Synchronous with backup workers combines the advantages of BSP and ASP by waiting only for N updates of the N+b workers, and thus ignoring the b slowest workers on every synchronization step (i.e., model update). Further strategies bound the maximum staleness (i.e., differences between the fastest and slowest worker) [Ho+13]. Recent work [YW+22] made a case for independent subnet training, where sparse disjoint subnets are assigned to the different workers which train these subnets locally. The coarse-grained iterative model aggregation and reassignment of independent subnets largely eliminates the need for synchronization and thus yields significant speedups, especially with many workers.

Reduced Communication Overhead: Given the broad applicability of the parameter server architecture and importance of fast communication, existing work extensively studied reducing the communication overhead. Commonly applied techniques include larger batch sizes [G+17] (i.e., fewer communications steps), more advanced optimizers [GKS18] (i.e., fewer model updates until convergence), decentralized training [L+17] (i.e., barriers among sub-groups of workers), prefetching and overlapping computation with communication [TF19] (e.g., layer-wise All-Reduce overlapped with backward pass computation), lossless and lossy compression of transferred data [S+14, J+18], sparse communication [R+22], different communication primitives, direct device communication, and even in-network aggregation [S+21].

BAGUA Demonstrator: In order to facilitate the exploration of combining such techniques for reduced communication overhead, we created the standalone BAGUA system [G+21]. BAGUA provides well-defined communication alternatives at extension hooks and thus, allows experimenting with combinations of optimization algorithms, communication primitives, and system relaxations. In detail, BAGUA includes parameter servers, collective operations like All-Reduce (MPI and NCCL), and decentralized learning; techniques for overlapping communication and computation, synchronous and asynchronous updates; and means of lossy compression and sparsification. A user specifies the network architecture, an optimizer, and a

BAGUA wrapper with additional configurations. BAGUA then provides a dedicated optimization framework for automatic scheduling and batching. This optimization framework applies overlapping communication and computation (via dynamic profiling), fusion and flattening of intermediates into continuous objects (batched communication), and hierarchical communication across nodes and multiple devices per node. The demonstrator artifact is available at https://daphne-eu.know-center.at/index.php/s/N6YJ3oScY7rS3tp and can be tested through the prepared tutorials at https://tutorials.baguasys.com/.

Scheduling in Different Environments: Additional prototypes focus on hybrid scheduling strategies in different environments such as in-database learning, function-as-a-service environments, and decentralized training strategies in large-scale heterogeneous networks. First, CorgiPile [XQ+24] introduces a hierarchical data shuffling strategy (for in-database learning) that avoids a full reshuffling before mini-batch training while maintaining good convergence rates of stochastic gradient descent. Second, LambdaML [JG+24] systematically explores and evaluates mini-batch training strategies in function-as-a-service (serverless) environments and finds that scheduling strategies need to be selected according to workload characteristics. Third, we introduce a new scheduling algorithm [YH+22] for training large foundation models in decentralized, heterogeneous, and low-bandwidth environments. This algorithm combines data-parallelism and pipeline parallelism, and computes cost-optimal groups of devices and communication schedules. These new scheduling strategies for different environments are a basis for a future holistic system support for alternative and extensible scheduling strategies for integrated data analysis pipelines.

2.5 Summary

The "Final Design of Scheduling Components and Extendability" details the DAPHNE system's scheduling framework, designed to efficiently manage tasks across distributed and shared-memory systems. This section introduces three key roles in scheduling decisions.

1) User-Driven Scheduling: while default scheduling settings are available for ease of use, expert users can adjust parameters such as task partitioning, thread allocation, and resource placement, tailoring performance without requiring deep scheduling expertise.

2) Compiler-Level Scheduling: At compile time, the DAPHNE compiler optimizes parallel execution by partitioning work, reordering tasks, and fusing operations. This approach improves efficiency by reducing redundant data movement and choosing suitable resources (e.g., CPU, GPU) for task execution. It ensures that tasks are structured to maximize memory locality and execution speed.

3) Runtime System Scheduling: The runtime system manages task execution using a vectorized approach, with a local scheduler for shared-memory processing and a distributed scheduler for multi-node coordination.

Dynamic scheduling techniques help manage workload distribution, while the modular design allows for future customizations, including support for new scheduling methods. The DAPHNE scheduling framework is highly extensible, enabling ongoing adaptation to meet specific performance goals and accommodate new developments in computational infrastructure.

3 Final DAPHNE Scheduler Prototype

This section presents the novel features in the final design of the DAPHNE scheduler. We summarize and provide examples on how to use the scheduling options of DAPHNE for the local runtime (Section 3.1) and for the distributed runtime (Section 3.2).

3.1 Local Runtime Scheduling Prototype Usage, Results, and Performance

The implementation of the entire **DAPHNE local runtime scheduler** involves numerous files and hundreds of lines of code. Detailed information and documentation, including extendibility guidelines, is available on the DAPHNE GitHub[3] repository.

The key files that are used for adding a new scheduling technique to the DAPHNE local scheduler are:

1. src/runtime/local/vectorized/LoadPartitioning.h
2. src/api/cli/daphne.cpp

The first file *LoadPartitioning.h* contains the implementation of the currently supported scheduling techniques, i.e., the current version of DAPHNE uses self-scheduling techniques to partition the tasks. In this file, the developer should change two things:

1. The enumeration that is called SelfSchedulingScheme. The developer will have to add a name for the new technique, e.g., MYTECH

*enum SelfSchedulingScheme { STATIC=0, SS, GSS, TSS, FAC2, TFSS, FISS, VISS, PLS, MSTATIC, MFSC, PSS, **MYTECH** };*

2. The function that is called getNextChunk(). This function has a switch case that selects the mathematical formula that corresponds to the chosen scheduling method. The developer has to add a new case to handle the new technique.

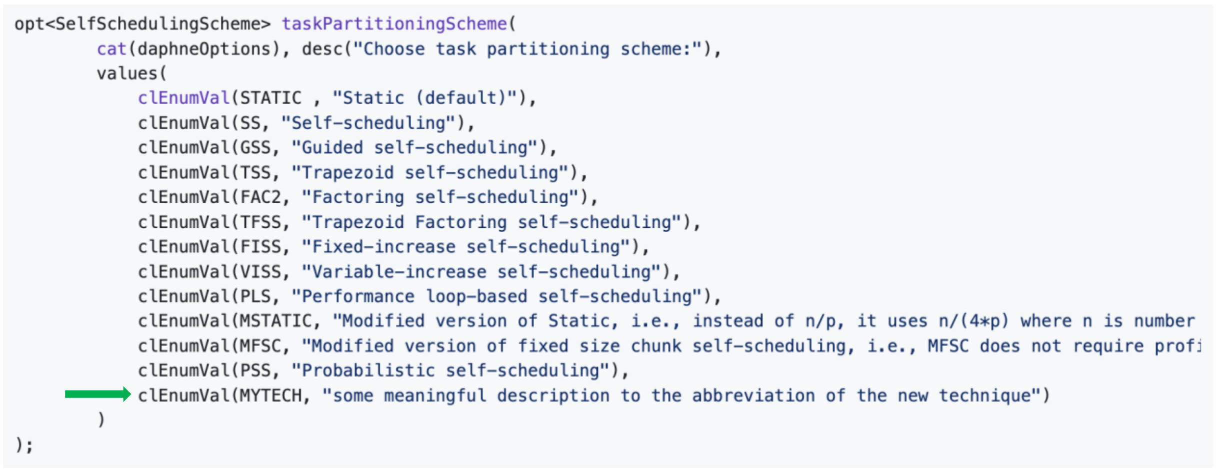
```
uint64_t getNextChunk(){
   //...
   switch (schedulingMethod){
       //...
       //Only the following part is what the developer has to add. The rest remains the same
       case MYTECH:{ // the new technique
           chunkSize= FORMULA;//Some Formula to calculate the chunksize (partition size)
           break;
       }
       //...
   }
   //...
   return chunkSize;
}
```

Enabling the selection of the newly added technique:

The second file daphne.cpp contains the code that parses the command line arguments and passes them to the DAPHNE compiler and runtime. The developer has to add the new

[3] https://github.com/daphne-eu/daphne/blob/main/doc/development/ExtendingSchedulingKnobs.md

technique as a vaild option. Otherwise, the developer will not be able to use the newly added technique. There is a variable called *taskPartitioningScheme* and it is of type *opt<SelfSchedulingScheme>*. The developer should extend the declaration of *opt<SelfSchedulingScheme>* as follows:



```
opt<SelfSchedulingScheme> taskPartitioningScheme(
        cat(daphneOptions), desc("Choose task partitioning scheme:"),
        values(
            clEnumVal(STATIC , "Static (default)"),
            clEnumVal(SS, "Self-scheduling"),
            clEnumVal(GSS, "Guided self-scheduling"),
            clEnumVal(TSS, "Trapezoid self-scheduling"),
            clEnumVal(FAC2, "Factoring self-scheduling"),
            clEnumVal(TFSS, "Trapezoid Factoring self-scheduling"),
            clEnumVal(FISS, "Fixed-increase self-scheduling"),
            clEnumVal(VISS, "Variable-increase self-scheduling"),
            clEnumVal(PLS, "Performance loop-based self-scheduling"),
            clEnumVal(MSTATIC, "Modified version of Static, i.e., instead of n/p, it uses n/(4*p) where n is number
            clEnumVal(MFSC, "Modified version of fixed size chunk self-scheduling, i.e., MFSC does not require profi
            clEnumVal(PSS, "Probabilistic self-scheduling"),
            clEnumVal(MYTECH, "some meaningful description to the abbreviation of the new technique")
        )
);
```

To use the new technique, one simply need to select "MYTECH" as the --partitioning command line option of DAPHNE. For more details, see the final prototype in the next subsections.

3.1.1 Usage

The following presents the steps to build and execute DAPHNE with scheduling options. The sequence of steps is similar to the steps in [D5.3].

Step 1 Download the snapshot from:

https://github.com/daphne-eu/daphne/archive/refs/tags/0.3.zip

```
unzip 0.3.zip
cd daphne-0.3
```

Step 2 Install dependencies:

Set up a Linux environment and install the software dependency versions specified in docs/GettingStarted.md. Other alternatives to build the DAPHNE prototype are described in docs/GettingStarted.md and include the use of containers, e.g., Docker and Singularity. The use of the provided containers is recommended to ensure that all required software dependencies and versions are correct.

Step 3 Build DAPHNE:

Within the daphne directory, run the build script. The first time DAPHNE is built; it may take ~30 minutes.

```
./build.sh
```

If the build fails, try to clean the build directory and rebuild DAPHNE as follows:

```
./build.sh --clean
./build.sh
```

Step 4 Check the Help menu of DAPHNE

After the installation is completed, one can access the help menu by executing the following command:

```
./bin/daphne --help
```

The output of the help command can be found below. The usability updates are highlighted in yellow and annotated with call-out boxes. The DAPHNE's help menu contains numerous other options which are not directly related to scheduling and were redacted to improve clarity.

```
OVERVIEW: The DAPHNE Prototype.
This program compiles and executes a DaphneDSL script.
USAGE: daphne [options] script [arguments]
OPTIONS:
Advanced Scheduling Knobs:
  --debug-mt                 - Prints debug information about the Multithreading Wrapper
  --grain-size=<int>            - Define the minimum grain size of a task (default is 1)
  --hyperthreading              - Utilize multiple logical CPUs located on the same physical CPU
  --num-threads=<int>           - Define the number of the CPU threads used by the vectorized execution engine
(default is equal to the number of physical cores on the target node that executes the code)
  --partitioning=<value>          - Choose task partitioning scheme:
    =STATIC                   -   Static (default)
    =SS                     -   Self-scheduling
    =GSS                     -   Guided self-scheduling
    =TSS                     -   Trapezoid self-scheduling
    =FAC2                     -   Factoring self-scheduling
    =TFSS                    -   Trapezoid Factoring self-scheduling
    =FISS                    -   Fixed-increase self-scheduling
    =VISS                    -   Variable-increase self-scheduling
    =PLS                     -   Performance loop-based self-scheduling
    =MSTATIC                   -   Modified version of Static, i.e., instead of n/p, it uses n/(4*p) where n is number of
tasks and p is number of threads
    =MFSC                    -   Modified version of fixed size chunk self-scheduling, i.e., MFSC does not require
profiling information as FSC
    =PSS                      -   Probabilistic self-scheduling
    =AUTO                     -   Automatic partitioning
  --pin-workers               - Pin workers to CPU cores
  --pre-partition              - Partition rows into the number of queues before applying scheduling technique
  --queue_layout=<value>          - Choose queue setup scheme:
    =CENTRALIZED             -   One queue (default)
    =PERGROUP                -   One queue per CPU group
    =PERCPU                  -   One queue per CPU core
  --vec                      - Enable vectorized execution engine
  --victim_selection=<value>       - Choose work stealing victim selection logic:
    =SEQ                      -   Steal from next adjacent worker (default)
    =SEQPRI                   -   Steal from next adjacent worker, prioritize same NUMA domain
    =RANDOM                   -   Steal from random worker
    =RANDOMPRI                -   Steal from random worker, prioritize same NUMA domain

...
```

The local DAPHNE runtime scheduler exposes 13 different partitioning techniques, 3 queue layouts, and 4 victim selections (only applicable when there are several queues). See Section 2.3.1 for a complete description of all options of the local DAPHNE scheduler.

One can use the following command and include --*timing* to record the execution time and set the number of workers with --*num-threads*. Here we start by using a single worker.

Step 5 Download the DaphneDSL Scripts and Input Matrix:

Download the archive containing the DaphneDSL scripts:

https://zenodo.org/records/14102956/files/D5.4.zip and decompress the archive in the daphne-0.3 directory.

```
unzip D5.4.zip
```

The DaphneDSL scripts need to operate on an input matrix. Download the input matrix (Wikipedia-20051105):

https://zenodo.org/records/14102956/files/wikipedia-20051105.tar.gz

Then, decompress the archive of the matrix, and set up the meta data:

```
tar -xzf wikipedia-20051105.tar.gz

echo '{"numRows":1634989,"numCols":1634989,"valueType":"f64","numNonZeros":19753078}' > wikipedia-
20051105/wikipedia-20051105.mtx.meta
```

Step 6 Execute DAPHNE:

```
./bin/daphne --vec --select-matrix-repr --timing --num-threads=1 --partitioning=STATIC ./D5.4/cc.daph
f=\"./wikipedia-20051105/wikipedia-20051105.mtx\"
```

The output of the command above should look similar to the following. Highlighted in **bold** is the execution time of the script.

```
{"startup_seconds": 0.021306, "parsing_seconds": 0.0030423, "compilation_seconds": 0.0499436,
"execution_seconds": 27.3668, "total_seconds": 27.4411}
```

One can also increase the total number of workers as follows:

```
./bin/daphne --vec --select-matrix-repr --timing --num-threads=6 --partitioning=STATIC D5.4/cc.daph
f=\"./wikipedia-20051105/wikipedia-20051105.mtx\"
```

The output below shows the increased performance with more workers:

```
{"startup_seconds": 0.0228854, "parsing_seconds": 0.00338405, "compilation_seconds": 0.0769501,
"execution_seconds": 16.5234, "total_seconds": 16.6266}
```

One can also change the partitioning technique, for example using AUTO instead of STATIC:

```
./bin/daphne --vec --select-matrix-repr --timing --num-threads=6 --partitioning=AUTO D5.4/cc.daph
f=\"./wikipedia-20051105/wikipedia-20051105.mtx\"
```

Which yields the following output:

```
{"startup_seconds": 0.0264353, "parsing_seconds": 0.00919016, "compilation_seconds": 0.061362,
"execution_seconds": 14.6002, "total_seconds": 14.6971}
```

3.1.2 Results

Figure 9 shows the performance of the different scheduling techniques offered by the local DAPHNE runtime scheduler for the execution of the Connected Components algorithm on the wikipedia-20051105 matrix. These experiments were conducted using 6 pinned workers (6 threads) and executed on an Intel Xeon E5-2640 v4 node with a total of 2 sockets, 10 cores per socket, and disabled hyperthreading. A single *CENTRALIZED* work queue (default) was considered. The results shown in Figure 9 are based on the execution of the following command repeatedly 5 times for each different "--partitioning" strategy. The dashed line corresponds to the mean execution time for the STATIC partitioning (the default partitioning for DAPHNE). The percentage represent the reduction in execution time compared to STATIC. Thus, for this particular experiment, MFSC performs 35% faster than STATIC. This highlights the importance of offering multiple scheduling techniques to users. The DAPHNE local scheduler is able to be adapted to cope with the underlying system and workload imbalance of application or inputs (e.g., matrices).

```
# The command below is a simple example and will not work if copied directly. We include it here simply to
illustrate the set of commands used to execute the experiment.


./bin/daphne --vec --select-matrix-repr --timing --num-threads=6 --partitioning={STATIC, GSS, FAC2, ... AUTO}
./D5.4/cc.daph f=\"./wikipedia-20051105/wikipedia-20051105.mtx\"
```

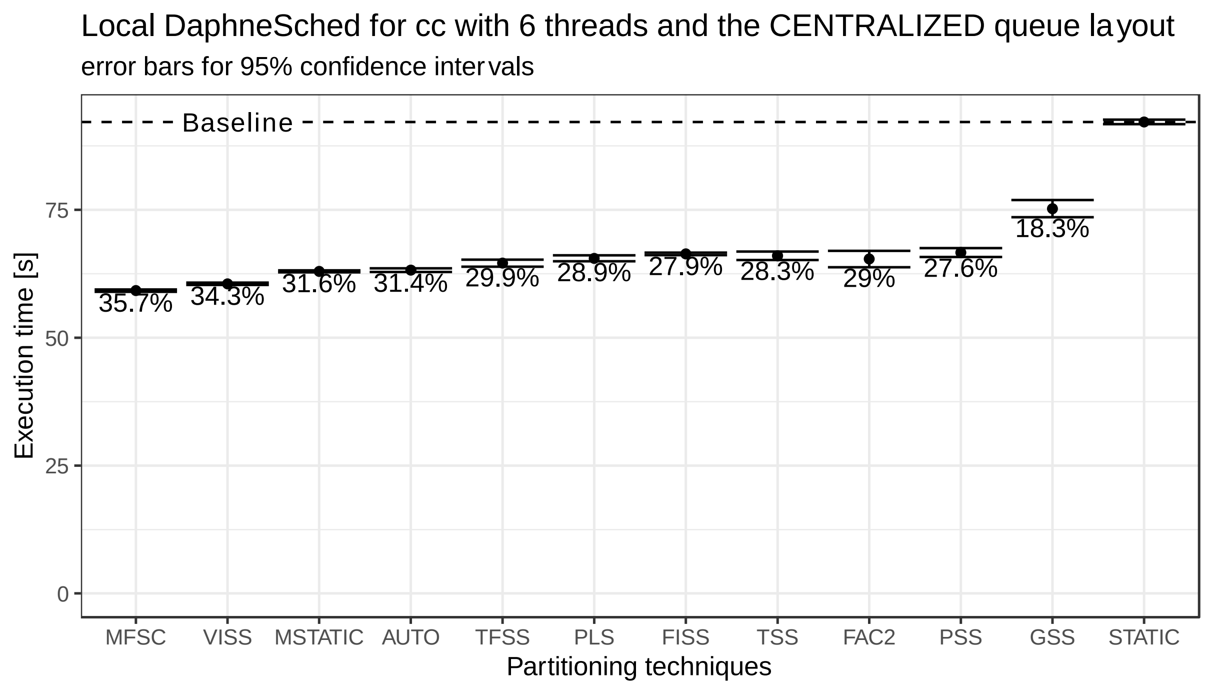


Figure 9 Execution time of the Connected Components (CC) application for several local DAPHNE runtime scheduler configurations.

3.2 Distributed Runtime Scheduling Usage, Results, Performance

The implementation of the whole DAPHNE **distributed** runtime scheduler also involves numerous files and hundreds of lines of code. Guidelines and documentation on extending the distributed DAPHNE scheduler can be found on the DAPHNE GitHub repository[4].

[4] https://github.com/daphne-eu/daphne/blob/e93c94345d3e6a79ff3bebde34c577c87a43278e/doc/development/ExtendingDistributedRuntime.md

3.2.1 Usage

Step 1 It is necessary to compile DAPHNE with MPI support

One can use the following command:

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

For this example, another DAPHNE script called *D5.4/nbody.daph* is used.

```
./bin/daphne --help
```

The prompt below highlights specific options related to the distributed DAPHNE runtime scheduler.

```
OVERVIEW: The DAPHNE Prototype.
This program compiles and executes a DaphneDSL script.
USAGE: daphne [options] script [arguments]
OPTIONS:

...

DAPHNE Options:

  --distributed                    - Enable distributed runtime

...

Distributed Backend Knobs:

  --dist_backend=<value>           - Choose the options for the distribution backend:
    =MPI                           -   Use message passing interface for internode data exchange (default)


...
```

For an execution with the distributed runtime, users must use the `--distributed` flag, as well as specifying the backend for the distributed runtime: either MPI or gRPC.

Step 2 Execute the *nbody.daph* DAPHNE script

The following command executes the example with the distributed runtime:

```
mpirun -np 2 ./bin/daphne --vec --num-threads=4 --partitioning=STATIC --distributed --timing --
dist_backend=MPI D5.4/nbody.daph nb_particles=100

{"startup_seconds": 0.109135, "parsing_seconds": 0.00602752, "compilation_seconds": 0.170672,
"execution_seconds": 29.7032, "total_seconds": 29.989}
```

This will start 2 MPI processes with one of them being the coordinator which will not execute any computation, and the other being a DAPHNE worker.

We can add more workers by increasing the number of MPI processes (from 2 to 4 below). In that case there will be one coordinator and 3 DAPHNE workers.

```
mpirun -np 4 ./bin/daphne --vec --num-threads=4 --partitioning=STATIC --distributed --timing --
dist_backend=MPI D5.4/nbody.daph nb_particles=100

{"startup_seconds": 0.119317, "parsing_seconds": 0.00564794, "compilation_seconds": 0.168974,
"execution_seconds": 31.8412, "total_seconds": 32.1352}
```

3.2.2 Results

To test the performance and scalability of the MPI distributed runtime of DAPHNE, we executed a N-body simulation on the VEGA supercomputer, where each node has an AMD Rome 7H12 with 2 sockets and 64 cores per socket. We can see on Figure 10 that as the number of nodes increase, the execution time of the simulation decreases. We also point out that the choice of the scheduling technique at the local DAPHNE runtime scheduler level has an influence on the total execution time. In these experiments, for the **local** DAPHNE runtime scheduler we consider the STATIC (default) and AUTO partitioning schemes while for the **distributed** scheduler the partitioning is STATIC.

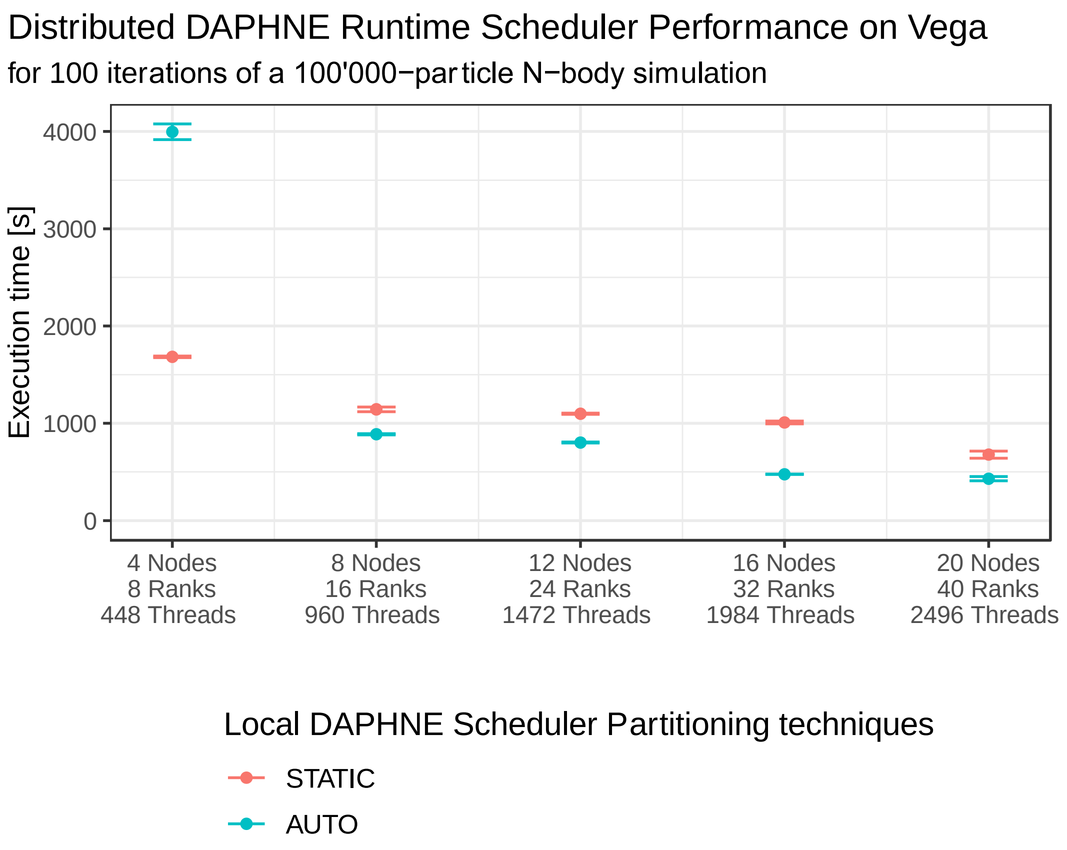


Figure 10 Performance of the DAPHNE MPI distributed runtime (N-body simulation).

3.3 Reproducibility

The development of the DAPHNE scheduler components (and DAPHNE as a whole) also considers reproducibility as a key aspect for the success of the project. ACM defines three levels of reproducibility[5]. The first level is "**Repeatability**", which means that one team of researchers is able to re-execute experiments on the same experimental setup. The next level is "**Reproducibility**", which means that a different team than the one that designed the experiments, is able to re-execute the experiments on the original experimental setup. The final

[5] https://www.acm.org/publications/policies/artifact-review-and-badging-current

level is "**Replicability**" where a different team is able to reach the same conclusions without access to the original experimental setup.

In the context of DAPHNE, and of this Work Package, we are mainly interested in the "Repeatability" and "Reproducibility" of our experiments. This means creating an experimental artifact that we can confidently share among members of the Work Package, the consortium, and Reproducibility reviewers of conferences.

3.3.1 Artifacts Description

In the case of DAPHNE, creating reproducible artifact mainly consists in a precise control of the software environment and of the DAPHNE version (i.e., git commit).

Concerning the software environment, DAPHNE provides two ways of setting it up: native installation of dependencies or use of containers.

In the `build.sh` building script, the third-party dependencies are downloaded and installed locally. The reproducibility of these installations is guaranteed by the fact that all the desired versions are correctly specified in the `software-package-versions.txt` file. Regarding containers, the Dockerfiles, which generate the containers, based themselves on the `build.sh` script. One potential reproducibility issue when using the native installation might come from the lack of control of the 2nd level dependencies to build the third-party dependencies of DAPHNE (e.g., the local version of the compiler). The Docker containers solve this issue by capturing these hidden dependencies at the build time of the container. However, rebuilding the same container image from the same Dockerfile in the future might not yield the same software environment as Dockerfiles rely on non-reproducible tools (e.g., apt). Hence, once the container has been built, a long-term storage of the result is crucial to achieve long-term reproducibility. Using DockerHub is a solution, but might only guarantee short- to middle-term storage, and thus reproducibility.

We provide an example of an artifact to reproduce Figure 9 of this deliverable following the guidelines on Zenodo: https://doi.org/10.5281/zenodo.14016947. Using Zenodo allow us to have long-term storage of the artifact. The Zenodo archive gathers several research objects. First the source code of the experiments. The source code of the experiments is also archived on Software Heritage with the complete and transparent history of the git repository: ***https://archive.softwareheritage.org/swh:1:dir:b834ba364fc85f64a8f0fc5218b13977e7fcd3d1;origin=https://bitbucket.org/unibasdmihpc/demonstrators;visit=swh:1:snp:b0551121df00380f9d4a96b95988cd0274838f48;anchor=swh:1:rev:c221210651ae8efe7cc764bde11166239a2c22eb*. The experiments will use the `daphne-dev` Docker image[6] to build and execute DAPHNE, as well as a Docker image for the analysis and plotting of the results. Uploading those Docker images to DockerHub provide an easy way for users to have access to the container, however, in the long-term, those images might be removed from DockerHub, or overwritten. Hence, the Zenodo archive also contains the Docker images. The daphne-dev image is produced with a classical Dockerfile recipe which might have some reproducibility issues if the recipe needs to be built in the future. The Docker image for the analysis and

[6] https://hub.docker.com/r/daphneeu/daphne-dev

plotting of the results has been built with Nix[7], which should provide a long term and exact rebuild of the Docker container in the future.

3.3.2 Artifacts Evaluation

One of our observations is that the process of running an artifact is often very manual and involves a lot of copy pasting, or executing fragile bash script. In this artifact for Figure 9, we use the GNU Make tool to provide a clear description (captured in the Makefile) of the workflow of the experiments (i.e., downloading the requirements, building DAPHNE, running the experiments, collecting the data, and plotting the results).

The workflow of the experiments requires to download some object from the internet (mostly from our Zenodo archive). When downloading an object, one should verify that the content of the downloaded object matches the expected content of the object to ensure the correctness of the download. To do so, we provide in the artifact, the sha256 hashes of all the objects that the artifacts might need to download. In the workflow, after each download the sha256 of the downloaded object is automatically verified to the expected one from the artifact. If the sha256 hashes differ, then the workflow will halt.

One complex aspect of the creation of artifact is their flexibility. By flexibility, we mean that if the system of the reproducer slightly differs from the one of the authors, then running the artifact might be much more cumbersome. In order to reduce the potential hurdles of the reproducer, we try to provide several means of installing DAPHNE (natively, with Docker, or with Singularity) and to execute the experiments (locally, or on a cluster managed by SLURM).

3.3.3 Reproducibility Badges

At the end of the artifact evaluation process, artifacts are awarded badges (left of Figure 11) to reflect their levels of reproducibility. Depending on the venue those badges differ slightly but encompass the same ideas.

In the case of ACM conferences, the main badges are the following: the “Artifacts Available” badge reward authors who did the efforts of publicly publishing their artifacts. The “Artifacts Evaluated” badge indicates that the artifact is well documented and contain all the required objects to attempt a reproduction of the results. Finally, the “Results Reproduced” badge indicates that the reproducibility reviewers managed to reproduce the main result from the paper.

Based on the results of our study on Artifact Descriptions of leading parallel and distributed conferences in 2023 [GC+24], we concluded that a crucial dimension of reproducibility is not currently covered by the current badges: the longevity of the artifact. One of the goals of reproducibility is to produce solid scientific work for future researchers to build upon, hence, if the artifact disappears or changes through time (e.g., source code of a dependency not available anymore), having the ”Results Reproduced” badge brings very little information about the artifact for the future researchers trying to use the scientific work.

Hence, there is a need to reward the “**Longevity**” dimension of artifacts, and thus proposed a new badge to integrate into the current badging system. We also proposed a grading system

[7] https://nixos.org

to award, at publishing time, the "**Artifacts Longevous**" badge (right of Figure 11) based on several criteria (how the artifact was shared, how the software environment was captured, and where the experiments were carried out).

Figure 11 Current ACM badges (left) and proposition for a new badge rewarding artifact that will pass the test of time (right).

3.4 Summary

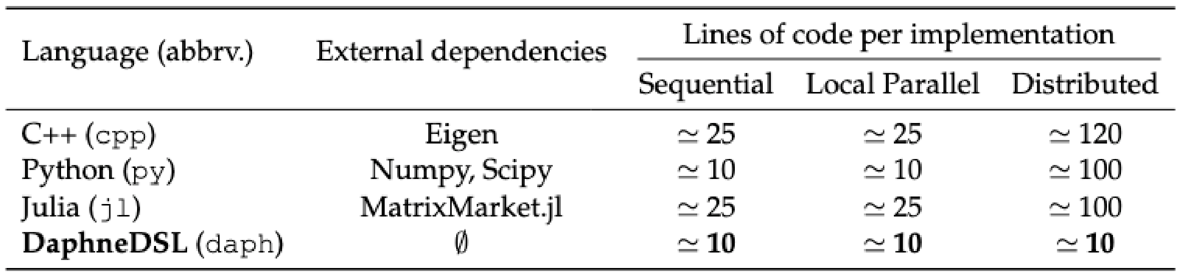
The "**Final DAPHNE Scheduler Prototype**" section presents a demonstration of usage of the local DAPHNE scheduler in Section 3.1, exhibiting the different options (partitioning techniques, queue layouts, victim selection, and number of threads options), in Section 3.2, we present the usage of the distributed DAPHNE scheduler, exemplified with the MPI backend. Section 3.3 summarizes our knowledge about reproducibility in the context of the DAPHNE project, and presents an example of artifact aiming for long-term reproducibility.

4 Conclusion

4.1 Discussion of Results and Comparison with other Frameworks

DAPHNE allows users to execute their IDA (Integrated Data Analysis) pipelines in a distributed setting without having to go through the hassle of integrating with libraries such as MPI as they would in different programming languages/frameworks. In this section, we present preliminary results comparing the scaling performances of an application executed with DAPHNE, Python, Julia, and C++, as well as the user's "effort" to integrate MPI in the sequential version of the application.

We compared the implementations of the Connected Components (CC) algorithm in C++, Python, Julia, and DaphneDSL for a sequential, single node parallel, and distributed executions. We collected the number of lines of codes as well as the number of required third-party libraries (required for linear algebra operations on sparse matrices and reading the MatrixMarket file format), and present them in Figure 12.



Language (abbrv.)	External dependencies	Lines of code per implementation		
		Sequential	Local Parallel	Distributed
C++ (cpp)	Eigen	$\simeq 25$	$\simeq 25$	$\simeq 120$
Python (py)	Numpy, Scipy	$\simeq 10$	$\simeq 10$	$\simeq 100$
Julia (jl)	MatrixMarket.jl	$\simeq 25$	$\simeq 25$	$\simeq 100$
DaphneDSL (daph)	$\emptyset$	$\simeq$ **10**	$\simeq$ **10**	$\simeq$ **10**

Figure 12 Characteristics of the implementation of the Connected Components algorithm in the considered languages for sequential, parallel on a single node, and distributed node executions.

For DAPHNE, no changes of the source code were required to execute in parallel or distributed setting. For the other languages, the linear algebra libraries can be parallelized by setting environment variables (e.g., OMP_NUM_THREADS), however, to execute the application on several nodes, the users need to integrate their application with libraries such as MPI, which increases the effort.

Figure 13 shows the strong scaling performance of the implementations of the CC algorithm on various node counts. Based on preliminary experiments, we used 20 MPI processes with 1 thread each on each node for C++, Julia, and Python. For DAPHNE, we used 1 MPI process per node with 20 threads per MPI process. We evaluated the default configuration of DAPHNE (STATIC with CENTRALIZED queue), and DAPHNE with AUTO scheduling on a CENTRALIZED queue, to highlight the potential gain to use a different scheduling technique. We observe that for small matrices (amazon0601), DAPHNE is outperformed by all other versions, with almost no scaling beyond 4 nodes. However, for larger and sparser matrices (wikipedia-20070206 and ljournal-2008), DAPHNE performs much better and even outperforms Python and Julia. C++ clearly outperforms all implementations.

We compared a DaphneDSL implementation for the Connected Components algorithm against Python, Julia, and C++ implementations along several dimensions: external dependencies, effort to adapt the code for parallel and distributed executions, and performance. The results demonstrate the value of DaphneSched's various scheduling options. Although DAPHNE does not yield the highest performance in a distributed setting compared to C++, it has the benefit of requiring no effort for seamlessly scaling applications across nodes with MPI and in certain circumstances outperforms Python and Julia implementations. Hence, DAPHNE positions itself as a trade-off between "Ease of Use" and "Scaling Performance".

The data and the analysis scripts are available on Zenodo[8].

[8] https://zenodo.org/records/11196525

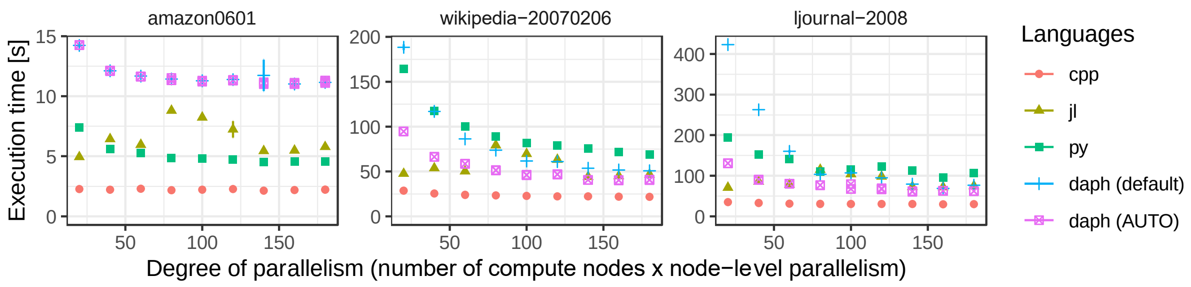


Figure 13 Strong scaling from 1 to 9 compute nodes. Inside a node, the work is parallelized with MPI for C++, Julia and Python, and with threads for DAPHNE. We execute DAPHNE with its default configuration (CENTRALIZED + STATIC) and with the AUTO scheduling technique and a CENTRALIZED queue to show the importance of scheduling on performance.

4.1.1 Results of Scheduling in the Local Runtime

The local runtime scheduler of the DAPHNE system was evaluated (results in Section 3.1.2, Figure 9) using the CC benchmark to test the different partitioning and schemes available. The results indicate that the choice of partitioning strategy significantly impacts the performance, particularly in scenarios with high variability in task execution times. Dynamic strategies such as FAC2 and FSC demonstrated reduced execution time for such irregular workloads compared to STATIC approaches. For regular workloads, simpler techniques like STATIC scheduling might still be viable as they incur lower scheduling overhead. Overall, the local runtime's flexibility in supporting a variety of scheduling techniques allows it to adapt to diverse application requirements, as evidenced by the performance improvements of up to 42% using FSC compared to STATIC.

The results in Figure 13 also show the potential of DAPHNE compared to other languages. Naturally, given the whole infrastructure of DAPHNE, the runtime overhead of the system is higher than lower-level languages such as C++. However, comparing DAPHNE against higher-level languages such as Python and Julia showed that performance is comparable while the development effort (in terms of lines of code) in DAPHNE is much smaller. Finally, the DAPHNE local runtime scheduler options are a unique feature that is **not** commonly available on other languages (unless explicitly implemented by external libraries or the user itself). As shown in Figure 13, the partitioning scheme AUTO from the local DAPHNE scheduler becomes extremely relevant even on distributed memory executions. Looking at the right plot of Figure 13, we can see that, despite C++, executions with AUTO outperform every other language/configuration.

4.1.2 Results of Scheduling in the Distributed Runtime

The distributed runtime scheduler extends the local scheduling capabilities to a multi-node setting, leveraging MPI or gRPC for inter-process communication. The evaluation focused on testing the scalability and efficiency of the distributed scheduler. The results in Sections 3.2.2 and 4.1 (Figures 10 and 13) showed a sub-linear speedup when increasing the number of distributed workers due to communication overhead and data distribution imbalances. The distributed DAPHNE scheduler still requires further optimization for enhanced performance.

The integration the of local and distributed schedulers ensures efficient task execution, demonstrating the scheduler's capability to manage both shared-memory and distributed-memory parallelism effectively.

4.2 Limitations and Suggestions for Overcoming Them

Despite its versatility, the DAPHNE scheduler still faces limitations that can affect performance in specific scenarios. First, the local runtime does not yet offer support for adaptive scheduling techniques. This can become a limitation in scenarios where there are system perturbations or heterogeneity. The implementation of adaptive partitioning techniques will require minor changes in the local runtime to allow the adaptation of task sizes during execution. Another limitation of the local DAPHNE scheduler is that for the usage of accelerators (e.g., GPUs), individual queues are created for CPUs and GPUs. This limits or complicates the process of load balancing work between CPU and GPU. The implementation of a common task queue and/or work stealing between queues for different types of devices can be done but requires significant changes in the DAPHNE runtime system.

The major limitation of the distributed DAPHNE runtime scheduler is that it relies on static partitioning and a centralized coordinator. The static partitioning limits the inter-node load balancing capabilities of DAPHNE while the centralized coordinator can become a bottleneck for large input data and large number of workers. Incorporating dynamic partitioning schemes and cross-node work-stealing mechanisms to the distributed DAPHNE runtime scheduler will enhance its load balancing capabilities. Also, the implementation of distributed coordination can eliminate the centralized coordinator bottleneck. The implementation of dynamic partitioning and work-stealing mechanisms can be achieved with the current design of the DAPHNE scheduler. Implementation of fully distribute coordination will require a different design.

4.3 Outlook

The work carried in Work Package 5 enables DAPHNE users to **achieve performance and scalability with no additional development effort**, while remaining in control of the scheduling configurations (partitioning techniques, queue layouts, victim selections). As there is not a one-fits-all scheduling configuration, it is crucial to yield control of this configuration to the users. DAPHNE's scheduling configurations could further be explored at runtime to find the best configuration for the user's application without having the user specifying any particular configuration. The use of exhaustive search and reinforcement learning methods is also to be considered.

5 Summary

This deliverable presents the final design and prototype of the DAPHNE scheduling components, showcasing an innovative framework for managing integrated data analytics pipelines that combine data management, high performance computing, and machine learning. It builds on the foundational architecture of the DAPHNE system, which incorporates user-driven flexibility, compiler-level optimizations, and robust runtime mechanisms to achieve efficient and scalable task execution.

The report begins by defining scheduling terminology and contextualizing the role of scheduling within the DAPHNE system. It highlights the interplay between user-configurable settings, compiler optimizations, and runtime adaptations that collectively ensure effective resource utilization and high performance. The core design of the scheduler emphasizes extensibility, enabling both default configurations and customizable strategies for diverse workload requirements.

The prototype implementation demonstrates the practical applicability of these concepts. It includes a local runtime scheduler optimized for shared-memory systems and a distributed runtime scheduler that coordinates work across multiple nodes. Experimental evaluations reveal significant performance gains achieved through various scheduling techniques, validating the effectiveness of the proposed methods.

While the current framework showcases significant advancements, limitations remain, particularly in scaling distributed coordination, and minimizing communication overheads. Addressing these challenges opens avenues for future enhancements, including decentralized scheduling architectures, adaptive partitioning strategies, and advanced hardware integration.

The deliverable concludes with a forward-looking perspective, emphasizing the potential for DAPHNE to support increasingly complex workloads and environments. By exploring advanced scheduling models, hybrid strategies, and real-time capabilities, the DAPHNE scheduler is poised to become a versatile and cutting-edge solution for large-scale data analytics workflows.

6 References

[BW91] Katherine M. Baumgartner and Benjamin W. Wah. Computer scheduling algorithms: past, present and future. Journal of Information Sciences, 57:319–345, 1991.

[Ull75] J. D. Ullman. NP-complete Scheduling Problems. Journal of Computer and System Sciences, 10(3):384–393, 1975.

[Leu04] Leung, Joseph YT, ed. Handbook of scheduling: algorithms, models, and performance analysis. CRC press, 2004.

[VA20] Versluis, Laurens, and Alexandru Iosup. (2020). "A Survey and Annotated Bibliography of Workflow Scheduling in Computing Infrastructures: Community, Keyword, and Article Reviews-- Extended Technical Report.". arXiv preprint:2004.10077.

[D+19] Deelman, Ewa, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. "The evolution of the Pegasus workflow management software." Computing in Science & Engineering, 2019.

[LTS+93] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In Proceedings of the international conference on parallel processing, 1993, pages 140–147.

[Luc92] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In Proceedings of the ACM Conference on Programming Language Design and Implementation, 1992, pages 200–211.

[Ban00] Banicescu, Ioana and Liu, Zhijun. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In Proceedings of the High performance computing Symposium, 2000, pages 122–129.

[Ban00] Banicescu, Ioana and Liu, Zhijun. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In Proceedings of the High performance computing Symposium, 2000, pages 122–129.

[EC23] Eleliemy, Ahmed and Ciorba, Florina. DaphneSched: A Scheduler for Integrated Data Analysis Pipelines. In Proceedings of the 22nd International Symposium on Parallel and Distributed Computing (ISPDC), 2023, pages 53–60

[PD22] P. Damme, et. al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. 2022.

[EC19] Ahmed Eleliemy and Florina M. Ciorba. Hierarchical Dynamic Loop Self-Scheduling on Distributed-Memory Systems Using an MPI+MPI Approach. In Proceedings of the International Parallel and Distributed Processing Symposium Workshops, 2019, pages 689‑697.

[CG17] Chen, Quan, and Minyi Guo. Task scheduling for multi-core and parallel architectures. Singapore: Springer Nature, 2017.

[WS81] Burton, F. Warren, and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In Proceedings of the 1981 conference on Functional programming languages and computer architecture, pp. 187-194. 1981

[PS14] S. Perarnau and M. Sato, “Victim Selection and Distributed Work Stealing Performance: A Case Study,” in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 659–668.

[CG17] Q. Chen and M. Guo, Task Scheduling for Multi-core and Parallel architectures, 1st ed. Springer Publishing Company, Incorporated, 2017.

[BBE+14] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, Yuanyuan Tian: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. IEEE Data Eng. Bull. 37(3): 52-62 (2014)

[EN16] R. Elmasri and S. B. Navathe. Fundamentals of Database Systems, Seventh Edition, Chapter 19. Pearson, 2016.

[HS82] T. C. Hu, M. T. Shing: Computation of Matrix Chain Products. Part I. SIAM J. Comput. 11(2): 362-373 (1982).

[HHH+21] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner: Simplicity Done Right for Join Ordering. CIDR 2021.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, Thomas Neumann: How Good Are Query Optimizers, Really? Proc. VLDB Endow. 9(3): 204-215 (2015)

[LRG+17] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, Thomas Neumann: Cardinality Estimation Done Right: Index-Based Join Sampling. CIDR 2017

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: TensorFlow: A System for Large-Scale Machine Learning. OSDI 2016: 265-283

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, Martin L. Kersten: MonetDB: Two Decades of Research in Column-oriented Database Architectures. IEEE Data Eng. Bull. 35(1): 40-45 (2012).

[BZN05] Peter A. Boncz, Marcin Zukowski, Niels Nes: MonetDB/X100: Hyper-Pipelining Query Execution. CIDR 2005: 225-237

[VL14] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In SIGMOD, 2014.

[BTR+14] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, Shivakumar Vaithyanathan: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. Proc. VLDB Endow. 7(7): 553-564 (2014)

[BRH+18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, Niketan Pansare: On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. Proc. VLDB Endow. 11(12): 1755-1768 (2018)

[BEK+17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah: Julia: A Fresh Approach to Numerical Computing. SIAM Rev. 59(1): 65-98 (2017)

[N11] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4(9): 539-550 (2011).

[A+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete

Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: TensorFlow: A System for Large-Scale Machine Learning. OSDI 2016: 265-283

[D+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, Andrew Y. Ng: Large Scale Distributed Deep Networks. NeurIPS 2012: 1232-1240

[GKS18] Vineet Gupta, Tomer Koren, Yoram Singer: Shampoo: Preconditioned Stochastic Tensor Optimization. ICML 2018

[G+17] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. CoRR abs/1706.02677, 2017

[G+21] Shaoduo Gan, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Jiawei Jiang, Binhang Yuan, Sen Yang, Ji Liu, Ce Zhang: BAGUA: Scaling up Distributed Learning with System Relaxations. Proc. VLDB Endow. 15(4): 804-813 (2021), https://github.com/BaguaSys/bagua

[Ho+13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, Eric P. Xing: More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. NeurIPS 2013: 1223-1231

[J+17] Jiawei Jiang, Bin Cui, Ce Zhang, Lele Yu: Heterogeneity-aware Distributed Parameter Servers. SIGMOD Conference 2017: 463-478

[J+18] Jiawei Jiang, Fangcheng Fu, Tong Yang, Bin Cui: SketchML: Accelerating Distributed Machine Learning with Data Sketches. SIGMOD Conference 2018: 1269-1284

[JG+24] Jiawei Jiang, Shaoduo Gan, Bo Du, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, Sheng Wang, Ce Zhang: A systematic evaluation of machine learning on serverless infrastructure. VLDB J. 33(2): 425-449 (2024)

[KMS20] Peter Kairouz, Brendan McMahan, and Virginia Smith. Federated Learning Tutorial. NeurIPS 2020. https://slideslive.com/38935813/federated-learning-tutorial

[Li+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, Bor-Yiing Su: Scaling Distributed Machine Learning with the Parameter Server. OSDI 2014: 583-598

[L+17] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, Ji Liu: Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. NeurIPS 2017: 5330-5340

[R+22] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, Volker Markl: NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. SIGMOD Conference 2022: 481-495

[S+10] Alexander J. Smola, Shravan M. Narayanamurthy: An Architecture for Parallel Topic Models. Proc. VLDB Endow. 3(1): 703-710 (2010)

[S+14] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, Dong Yu: 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. INTERSPEECH 2014: 1058-1062

[S+21] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, Peter Richtárik: Scaling Distributed Machine Learning with In-Network Aggregation. NSDI 2021: 785-808

[TF19] Google: Inside TensorFlow: tf.distribute.Strategy, 2019,

https://www.youtube.com/watch?v=jKV53r9-H14

[XQ+24] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cédric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, Jieping Ye, Ce Zhang: Stochastic gradient descent without full data shuffle: with applications to in-database machine learning and deep learning systems. VLDB J. 33(5): 1231-1255 (2024)

[GC+24] Quentin Guilloteau, Florina M Ciorba, Millian Poquet, Dorian Goepp, Olivier Richard. Longevity of Artifacts in Leading Parallel and Distributed Systems Conferences: a Review of the State of the Practice in 2023. REP 2024 - ACM Conference on Reproducibility and Replicability, ACM, Jun 2024, Rennes, France. pp.1-14

[BE+16] Matthias Boehm, Alexandre V. Evfimievski, Niketan Pansare, Berthold Reinwald: Declarative Machine Learning - A Classification of Basic Properties and Types. CoRR abs/1605.05826 (2016)

[D3.1] DAPHNE: D3.1 Language Design Specification, EU Project Deliverable

[D3.4] DAPHNE: D3.4 Compiler Design and Overview, EU Project Deliverable

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

[D5.1] DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks, EU Project Deliverable

[D5.2] DAPHNE: D5.2 Prototype of Pipelines and Task Scheduling Mechanisms, EU Project Deliverable

[D5.3] DAPHNE: D5.3 Improved Prototype of Pipelines and Task Scheduling Mechanisms, EU Project Deliverable

[D8.1] DAPHNE: D8.1 Initial Pipeline Definition of all Use Cases, EU Project Deliverable