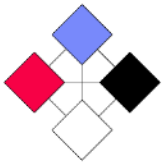
D5.1 Scheduler Design for Pipelines and Tasks



DAPHNE

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

Version 2.1
PUBLIC

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

Document Description

This report describes the scheduling context and decisions in the DAPHNE system. It also offers a detailed description of the preliminary design of the DAPHNE scheduler for pipelines and tasks. This design reflects the continuous discussion between all DAPHNE partners, especially those from WP2 (System Architecture), WP3 (DSL Abstractions and Compilation), WP4 (DSL Runtime and Integration), and WP5 (Scheduling and Resource Sharing). Extensibility is at the heart of the DAPHNE scheduler and we have carefully chosen a set of scheduling strategies and techniques in this initial design to support it. We designed the scheduler to also be extendable with other new scheduling techniques, including user-defined strategies. To make this report self-contained, we start by defining the relevant scheduling terminology, then describe the scheduling-related components from the DAPHNE system architecture, with focus on the Tiled execution engine. The last parts of this report present the initial design of scheduling components and techniques considered by the DAPHNE compiler and runtime system. As the DAPHNE project continues, this initial design will incrementally evolve as needed. Updates to this initial design will be accordingly reflected in a refined scheduler design in future deliverables D5.2, D5.3, and D5.4 (planned for M21, M36, and M48, respectively).

D5.1 Scheduler Design for Pipelines and Tasks			
WP5 - Scheduling and Resource Sharing			
Type of document	R	Version	2.1
Dissemination level	PU		
Lead partner	UNIBAS		
Author(s)	Florina M. Ciorba (UNIBAS), Patrick Damme (KNOW), Ahmed Eleliemy (UNIBAS), Vasileios Karakostas (ICCS), Gabrielle Poerwawinata (UNIBAS)		
Reviewer(s)	Marius Birkenbach, Wolfgang Lehner		

Revision History

Version	Item	Comment	Author / Reviewer
V1.0	Outline	Initial write-up of the report outlines	Florina Ciorba Ahmed Eleliemy Gabrielle Poerwawinata
V1.7	Updated content	Extended Section 4, and refined Sections 1, 2, 3, and 4	Florina Ciorba Ahmed Eleliemy Vasileios Karakostas
V1.11	Updated content and added more content	Refined Sections 1.3.2, 2, and 3, extended elaboration on compiler decisions about task-level parallelism in Section 3, added references (in Sections 3 and References), addressed comments by co-authors	Patrick Damme
V2.1	revision	Addressing comments of Wolfgang Lehner and Marius Birkenbach	Florina Ciorba Ahmed Eleliemy

Table of Contents

Table of Figures

List of Tables

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
NUMA	Non-Uniform Memory Access policy
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 (DaphneDSL, Compiler, and Runtime) is designed for enabling 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 codes are commonly executed on distributed-memory systems that include heterogeneous resources (Figure 1). These distributed-memory systems range from traditional HPC clusters with a shared-disk setup to shared-nothing systems. Scheduling is a cornerstone to achieving various performance targets, the most common being minimizing application execution time, increasing resource utilization, and increasing

computing throughput. The DAPHNE system is not an exception and scheduling performs an essential role.

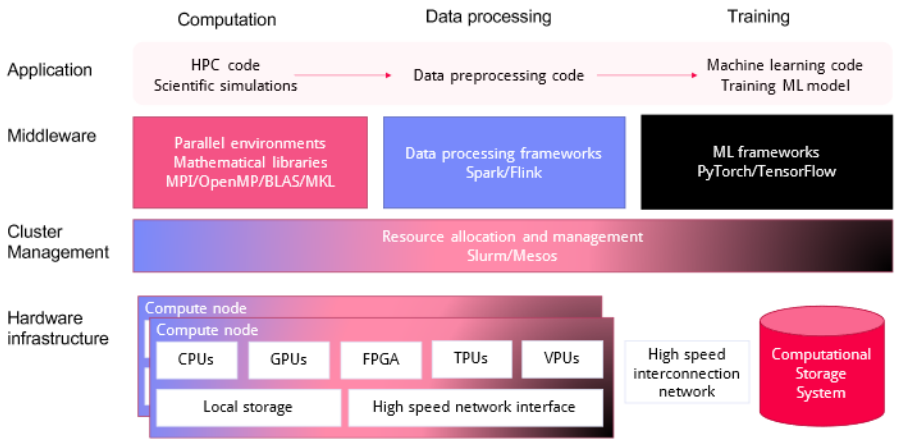


Figure 1 Ecosystem for an integrated data analytics pipeline [IPE+21]

Scheduling refers to mapping units of work to computing resources over a specific period of time [BW91] [Ull75]. Scheduling solutions fall into several classes (online, offline, optimal, heuristics, etc.). Scheduling involves various decisions regarding work partitioning, work assignment, and execution ordering [BW91]. Scheduling decisions are taken at different levels of hardware parallelism in the HPC systems (core, node, and system). Scheduling techniques range from static to dynamic depending when those scheduling decisions are taken.

Table 1 shows the different scheduling decisions taken by a DAPHNE user, the DAPHNE compiler and the runtime system. We distinguish scheduling by the local runtime system from that in the distributed runtime system by highlighting specific implementation details. We use Table 1 to guide our description of the techniques employed by the compiler and the runtime system, their associated scheduling decisions, and optimization goals.

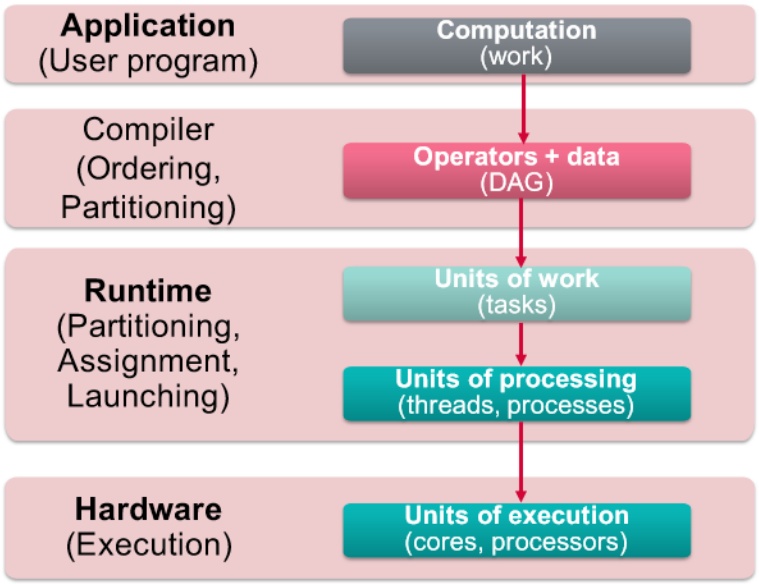


Figure 2 From a DAPHNE program to parallel execution

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 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).

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

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, 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.

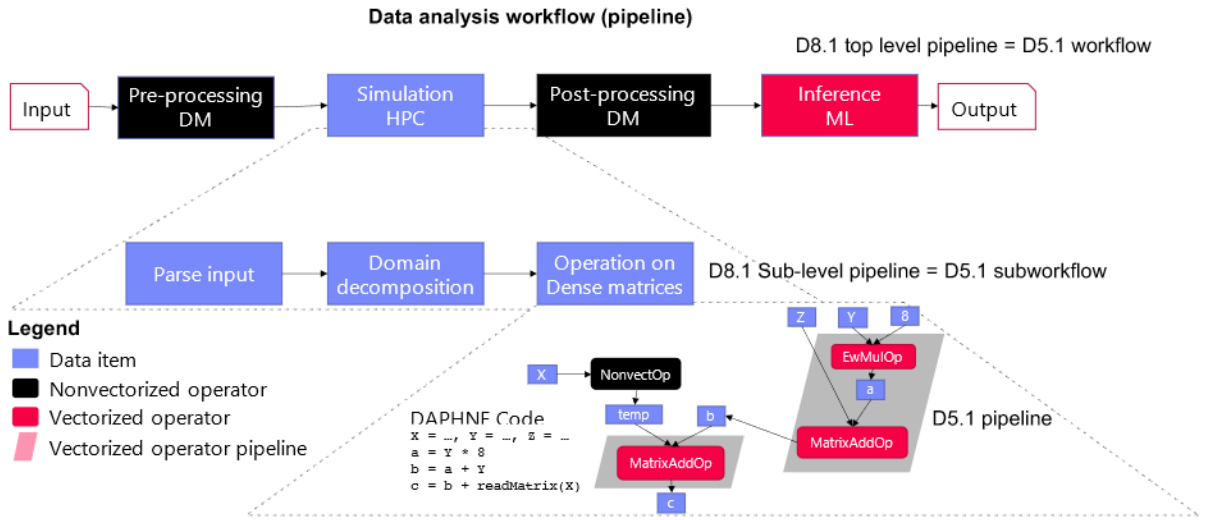


Figure 3 Data analysis workflow and the associated terminology at various levels

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 a piece of 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. 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.

1.3.1 Scheduling by the User

DAPHNE users can take certain scheduling decisions. For instance, using DaphneDSL, a user may use ParFOR (described in Section 2) to express a certain loop as a loop without data/control dependencies, or a DOALL loop [DP91]. The user may also take decisions about the number of threads, the scheduling technique to be used, etc. By providing certain configuration options to execute a DAPHNE program, the user can generally influence all scheduling decisions at the application level.

1.3.2 Scheduling in the Compiler

The compiler decides how to partition the DAG of operators and their associated data into a set of pipelines. Furthermore, the compiler decides whether these pipelines are vectorizable as well as the best order to execute these pipelines on the target devices.

1.3.3 Scheduling in the Runtime System

When executing a vectorized pipeline, the runtime system takes scheduling decisions similar to those of the compiler. It decides the partitioning of the input data. The data partitions and their associated (vectorized) operators form **tasks**. The size of data partition determines the *granularity* of the **vectorized tasks**. The runtime system also takes decisions regarding mapping (assignment) of such tasks onto the available hardware execution units.

The **Tiled Execution Engine** compiles operator pipelines of data frames and matrices across hardware devices. The pipelining allows fine-grained operator fusion [PD22]. A vectorized operator pipeline has multiple inputs, outputs, and intermediate representation (IR) bodies similar to LLVM loops. An input matrix can be federated across the memory of hardware devices, such as CPUs, GPUs, and FPGAs. The DAPHNE vectorized execution engine works similarly to Morsel [KD19][VL14], by creating vectorized tasks for aligned rows or columns of input matrix partitions and appending them into a single or multiple task queues. Each of the vectorized tasks comprises of its input data, an operator pipeline (DAG) with specific input data such as scalar, a matrix row, or a matrix tile. Worker threads process the tasks in the queue and combine the results. Vectorized execution is also integrated with computational storage, for instance, on near-SSD CPUs or FPGAs, to support asynchronous I/O and subsequent computation pipelines for the task queues [PD22].

2 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 administrator), as described in Sections 3 and 4. However, expert users may optionally configure the scheduling behavior of some DaphneDSL program manually. Available tuning knobs include, e. g., 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.

These scheduling-related parameters can be provided at different levels of granularity. Configuration files and command line arguments can be used to set the scheduling parameters globally for all DaphneDSL files or one specific DaphneDSL file, respectively. Furthermore, the extensibility mechanisms of DAPHNE can be used to influence scheduling.

The DaphneDSL will provide means to set locally certain scheduling-related parameters for parts of the DaphneDSL source code via dedicated built-in functions (for more details, see Deliverable D3.1). For instance, take a *parallel for* loop construct, called ParFOR, allowing the explicit expression of task-level parallelism. With a ParFOR loop, the user may specify the degree of parallelism, the task partitioning and result merge methods, and the task size. Furthermore, the user may assign individual operations and data objects to certain devices, such as GPUs.

Sideways access into DaphneIR allows expert users to directly modify the internal representation to influence compiler decisions in a fine-grained way.

3 Scheduling in 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. 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 will be provided in Deliverable D3.4 (due in M36) which will include the compiler design and overview; here we only provide a brief overview and highlight *how compiler decisions relate to scheduling*.

3.1 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.

3.2 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) 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 4), 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.

3.3 Decisions about Data-level Parallelism

The DAPHNE compiler performs intra- and inter-procedural analyses [BBE+14] to estimate the dimensions, 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 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**.

3.4 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) 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.

3.5 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.

3.6 Placement

By means of configuration (for more details, see Deliverable D3.1), 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.

4 Scheduling in the Runtime System

he tiled execution engine is one of the earliest design decisions that we considered in the initial DAPHNE system architecture. The tiled 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.

Dynamic loop self-scheduling (DLS) techniques have been devised to schedule loop iterations that have no data dependencies (or for which dependencies have been resolved through various loop transformation techniques during compilation) and **each loop iteration** is considered to be a **task**. DLS techniques divide the tasks that describe the DAPHNE program into chunks and self-schedule these chunks to the available software units of processing (processes, threads) following the self-scheduling principle [WS81].

The DAPHNE runtime system then decides the assignment of those software units of processing to the hardware units of execution (CPUs, GPUs, FPGAs). As shown in Table 1, two scheduling decisions are taken by a DLS technique: work partitioning and work assignment.

4.1 Work Partitioning

As mentioned in Section 1, **work partitioning** refers to the partitioning of the work into units of work (or tasks) according to a certain granularity (fine or coarse, equal or variable). Tasks can be vectorized tasks that means work partition is about data partitioning, while when tasks are not vectorized work partition is about functional parallelism. In any case, deciding the granularity of work to be assigned to individual execution units to balance the execution progress among them.

The work granularity can be controlled by the two strategies illustrated in Figure 4.

For instance, Figure 4 shows matrices A and B with n rows ($r_0..r_{n-1}$). The work in this case is adding the two matrices. The work can be partitioned into either t_n or t_m tasks ($m<n$), which are then mapped to c_n or c_m chunks, and later assigned in a sender- or receiver-initiated fashion to the available threads for execution.

For design simplicity, we will use in DAPHNE the second granularity control strategy to avoid creating additional level of abstraction, i. e., tasks of variable size are considered chunks.

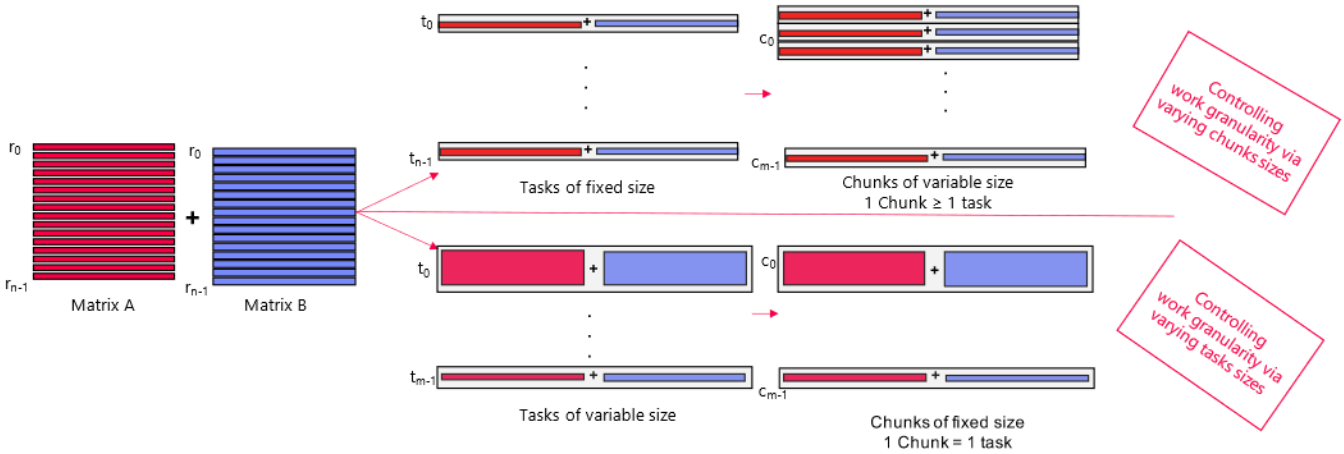


Figure 4 Strategies for controlling work granularity

4.1.1 Design of the Work Partitioner

The DAPHNE runtime system contains a component that partitions the tasks. This component is illustrated in Figure 5, and denoted as work (or task) partitioner. It has three interface points, each implemented as functions. The first initializes the load partitioner (initialize), the second gets the count of remaining tasks (getNumRemainingTasks), and the third gets a task for execution (getTask).

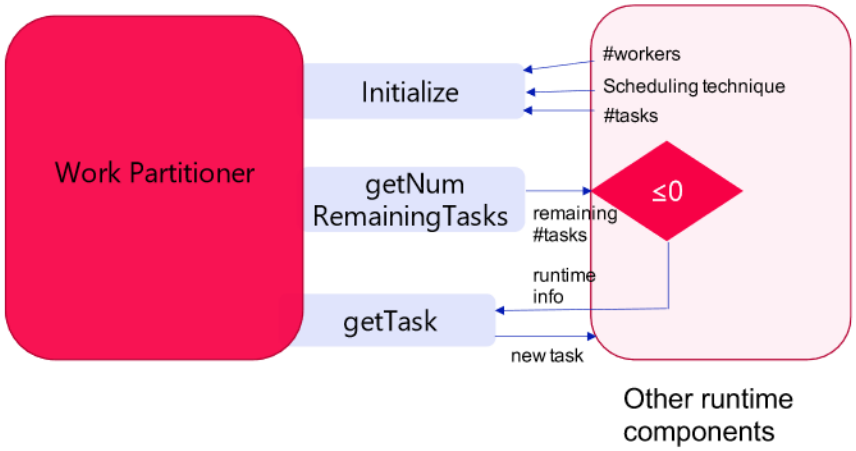


Figure 5 The work partitioner and its iterative design

The work partitioner is designed to be used in an **iterative** fashion. Since task partitioning is *separate* from task assignment, one may consider the iterative aspect in the design as a potentially unrequired level of complexity. However, as discussed earlier in Section 1.2, the adaptive scheduling techniques require a live feedback mechanism to refine scheduling decision based on information only available during execution. Therefore, the iterative work partitioning design is required to support adaptive DLS techniques.

We have identified a number of DLS techniques to be supported in the DAPHNE system, classified in Figure 5 and described below. The notation used to describe the chosen scheduling techniques is summarized in Table 2.

Table 2 Notation for describing scheduling techniques

Symbol	Description
N	Total number of tasks
P	Total number of hardware execution units
S	Total number of scheduling rounds
i	Index of current scheduling round, 0 ≤ i ≤ S − 1
R_i	Remaining loop iterations after i-th scheduling step
L	A DLS technique
K_0^L	Size of the initial chunk of a scheduling technique L
K_{S-1}^L	Size of the last chunk of a scheduling technique L
K_i^L	Chunk size calculated at scheduling step i of a scheduling technique L
μ	Mean of the loop iterations' execution times
σ	Standard deviation of the task execution times
$\Phi(P)$	Probability of having P workers available

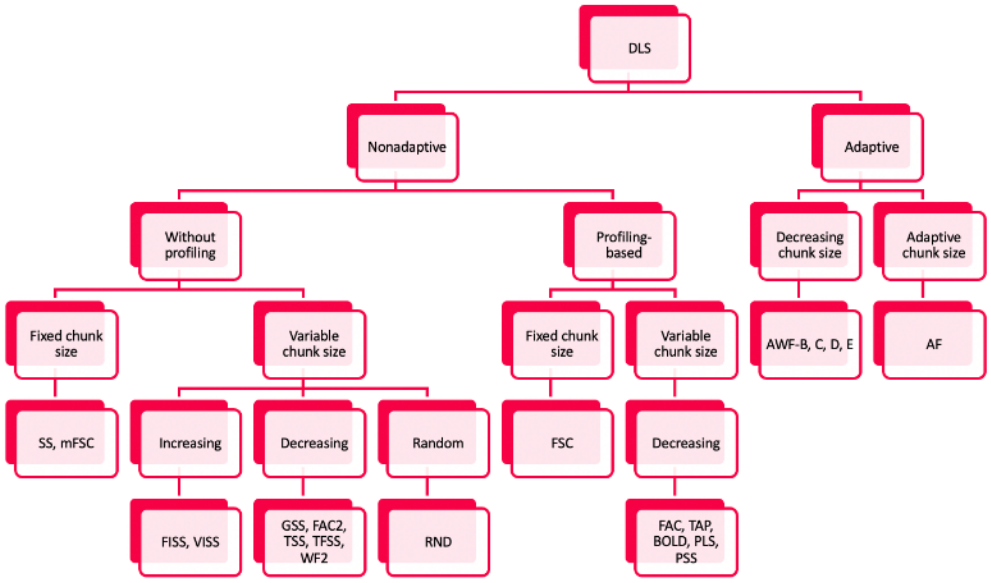


Figure 6 Work partitioning during execution with DLS techniques

4.1.1.1 Set of Implemented DLS Techniques (Nonadaptive without Profiling)

- Block static (STATIC) [LTS+93].; it is a straightforward technique that divides tasks into P chunks of equal size as follows $K_i^{STATIC} = \frac{N}{P}$.
- Dynamic self-scheduling (SS) [PK87] technique where the chunk size is always one task as follows $K_i^{SS} = 1$.
- Fixed size self-scheduling (FSC) [KW85] assumes an optimal chunk size that achieves a balanced execution of loop iterations with the smallest overhead. FSC considers the variability in iterations' execution time and the scheduling overhead of assigning task to be known before applications' execution. To calculate such an optimal chunk size, FSC considers the variability in tasks' execution time and the

scheduling overhead of assigning tasks to be known before applications' execution. A practical implementation of FSC is called mFSC and does not require such information as follows $K_i^{FSC} = \frac{\sqrt{2} \cdot N \cdot h}{\sigma \cdot P \cdot \sqrt{\log P}}$.

- Guided self-scheduling (GSS) [PK87] assigns decreasing chunk sizes to balance loop executions among all execution units as follows $K_i^{GSS} = \frac{R_i}{P}$.
- Trapezoid self-scheduling (TSS) [TN93] assigns decreasing chunk sizes similar to GSS. However, TSS uses a linear function to decrement chunk sizes as follows $K_i^{TSS} = K_{i-1}^{TSS} - \left\lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \right\rfloor$, where $S = \left\lceil \frac{2 \cdot N}{K_0^{TSS} + K_{S-1}^{TSS}} \right\rceil$, and $K_0^{TSS} = \left\lceil \frac{N}{2 \cdot P} \right\rceil, K_{S-1}^{TSS} = 1$.
- Factoring (FAC) [FHSF92] schedules the loop iterations in batches of equally-sized chunks. FAC evolved from comprehensive probabilistic analyses, and it assumes prior knowledge about μ and σ. Another practical implementation of FAC denoted, FAC2, assigns half of the remaining loop iterations for every batch as follows

$$K_i^{FAC2} = \begin{cases} \left\lceil \frac{R_i}{2 \cdot P} \right\rceil, \text{if } i \bmod P = 0, \\ K_{i-1}^{FAC2}, \text{ otherwise.} \end{cases}$$

- Trapezoid factoring self-scheduling (TFSS) [CAB+01] combines characteristics of TSS and FAC. It schedules loop iterations in batches of equally-sized chunks. Similar to FAC, TFSS schedules loop iterations in batches of equally-sized chunks. Every batch in TFSS decrease linearly, similar to chunk sizes in TSS

$$K_i^{TFSS} = \begin{cases} \frac{\sum_{j=i}^{i+P-1} K_j^{TSS}}{P}, & \text{if } i \bmod P = 0, \\ K_{i-1}^{TFSS}, & \text{otherwise.} \end{cases}$$

4.1.1.2 Other Techniques (Nonadaptive with Profiling and Adaptive)

- PLS divides the loop into two parts. The first part is scheduled statically, while the second part is scheduled dynamically using GSS. The static workload ratio (SWR) is used to determine the amount of the iterations to be statically scheduled. PLS uses a performance function to statically assign parts of the workload to each processing element (PE) based on the PE's speed and its current CPU load [SYT07]. In this work, all PEs are assumed to have the same load during the execution [SYT07].

$$K_i^{PLS} = \begin{cases} \frac{N \cdot SWR}{P}, \text{if } R_i > N - (N \cdot SWR) \\ K_i^{GSS}, \text{ otherwise.} \end{cases} \text{, where}$$

$$SWR = \frac{\text{minimum iteration execution time}}{\text{maximum iteration execution time}}.$$

- PSS schedules the number of iterations allocated to an idle processor based on the number of remaining iterations and the number of processors expected to be available in the future [MG06]:

$$K_i^{PSS} = \left\lceil \frac{R_i}{1.5\ \Phi(P)} \right\rceil.$$

- AWF executes variably-sized chunks of a given batch according to its relative weight. The weight for each of the processing elements will be updated during execution at the end of each time step based on the performance of the processor [FHSU+96]. AWF variants such as AWF-B and AWF-C relaxed the constraint of updating the processor weights from at the end of each time-step to at every batch and chunk [CB08]. Other variants such as AWF-E and AWF-D works similarly to the AWF-B and AWF-C with scheduling overhead consideration in measuring the relative weights.

- AF is an adaptive DLS technique based on FAC. In comparison to FAC, AF learns the μ and σ for each computing resource during application execution [BAN00]. The continuous updates of loop iteration execution μ and their standard deviation σ adapt the chunk size during execution.

$$K_i^{AF} = \frac{D+2\cdot E\cdot R_i-\sqrt{D^2+4\cdot D\cdot E\cdot R_i}}{2\mu_{p_i}}, \text{ where } D = \sum_{p_i}^{P} \frac{\sigma_{p_i}^2}{\mu_{p_i}},\ E = \left(\sum_{p_i=1}^{P} \frac{1}{\mu_{p_i}}\right)^{-1}.$$

4.2 Work Assignment

Work assignment refers to mapping (or placing) the units of work (or tasks) onto individual units of execution (processes or threads) (see Section 1). All formulas mentioned in Section 4.1 are concerned with determining the best granularity of the tasks (chunk size) to maximize application performance. Work assignment is *independent* of task granularity, 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. A **work queue** refers to a standard thread safe data structure (a double ended queue or dequeue for short) that applies the **first-in first-out** access policy.

4.2.1 Work Sharing across Homogeneous Workers (Centralized Work Queue)

In **work sharing**, all tasks are created and stored in one **centralized work queue** as shown in Figure 7. Once a worker thread becomes free and available it obtains a new task to execute from this centralized work queue. Work sharing 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. Work sharing has the following **disadvantages**: 1) when the number of workers increases, access to the centralized work queue becomes a synchronization bottleneck that may negatively impact performance and 2) is not data locality-aware.

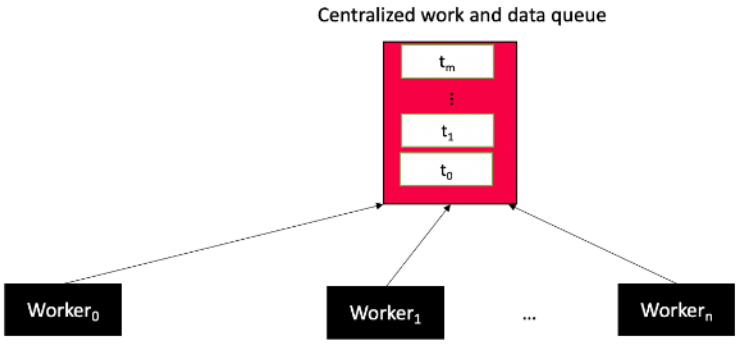


Figure 7 Work assignment using work sharing following the self-scheduling execution principle

4.2.2 Work Stealing across Homogeneous Workers (Multiple Work Queues)

In **work stealing**, each worker has a local work queue as shown in Figure 7. Worker threads obtain tasks from the individual work queues, and only when the local work queue is out of tasks, a worker steals from another worker's work queue. Work stealing has the following **advantages**: 1) relieves the contention associated with concurrent access to a centralized work queue and 2) is data locality-aware. Work stealing has the following **disadvantages**: 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.

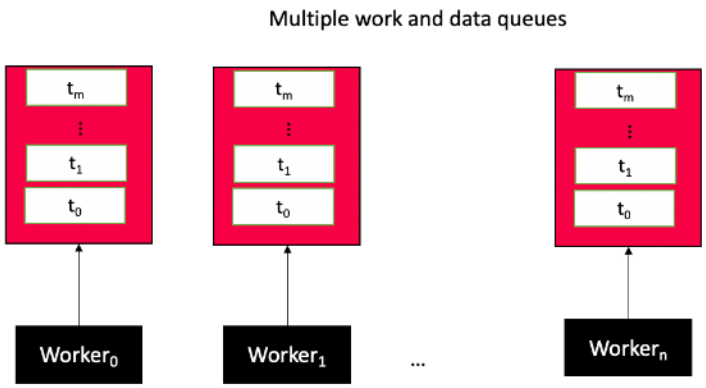


Figure 8 Work assignment using work stealing following the self-scheduling execution principle

4.2.3 Work Sharing and Work Stealing across Heterogeneous Workers (Code Generation)

The DAPHNE system is designed to support heterogeneous workers, i. e., workers are software units of execution (processor or threads) which abstract existing hardware units of execution, including, but not limited to, CPUs, GPUs, FPGAs, and computational storage devices. For **work partitioning**, the DLS techniques that follow the **self-scheduling principle**, i. e., weighted factoring (WF) [FHSU+96], adaptive factoring (AF) [BAN00], and adaptive weighted factoring (AWF) [CB08], were devised for scheduling on heterogeneous workers. For **work assignment**, DAPHNE compiles a device-specific kernel for different workers. The compiler generates codes that handles fine-grained specialization and exploitation of heterogeneous devices. Therefore, we initially support **work sharing** at the device level, i. e., workers of the same type share a centralized work queue. **Work stealing** is also supported among workers of the same type.

4.3 Work Ordering

Scheduling in the DAPHNE runtime system is **currently not concerned with the order in which the tasks are executed**. Since we rely on the tiled execution engine in the initial design, tasks within a vectorized pipeline have no dependencies and thus can be executed in any order. Furthermore, pipelines are currently launched one-by-one. Therefore, the execution order of vectorized tasks only depends on the task order in the queue (FIFO). In the future, we plan to relax this constraint and consider the parallel and out-of-order execution of operations based on the DAG information that the DAPHNE compiler provides.

4.4 Local and Distributed Scheduling

The DAPHNE runtime system is designed to execute on both **local** and **distributed** systems. Thus, it includes two different schedulers, a **local scheduler** that is responsible for scheduling tasks on the resources of a compute node that may consist of CPUs, GPUs, FPGAs, and computational storage devices, and a **distributed scheduler** that is responsible for scheduling tasks on a cluster of compute nodes. Both types of schedulers take decisions regarding **work partitioning** and **assignment**. For instance, the local scheduler concurrently maps multiple tasks onto the individual resources of a compute node–taking into consideration the NUMA topology and the performance interference that may arise due to resource sharing.

Both schedulers (local and distributed) may rely on DLS techniques for work partitioning, and both can use work queues and the self-scheduling principle for work assignments. However, each scheduler considers different parameters regarding the behavior of the application tasks, the computational capabilities of the hardware resources, the memory hierarchy and data locality, and the topology of the hardware resources and the compute nodes. Consequently, these schedulers may employ different scheduling techniques at the same time and exploit various scheduling structures.

For instance, for CPU workers that reside on the same compute node, shared-memory node, accessing a *local centralized work queue* introduces less overhead than accessing

globally distributed work queues that may be hosted by different compute nodes. The latter access will have to go over network and use data distribution primitives (for more details, see Deliverable 4.1) to bring the new tasks. In addition, a local scheduler may use fine-grained tasks to minimize load imbalance, while a distributed scheduler may use coarse-grained tasks to minimize communication associated with work assignment and avoid frequent access to distributed work queues.

4.4.1 Device-level Scheduling (Local Scheduling)

The DAPHNE local scheduler is also responsible for selecting the target device, e. g., CPU, GPU, FPGA, or computational storage, to execute individual tasks. Selecting such devices offers the potential for improving different metrics, such as reduced latency, increased throughput, increased resource utilization, and increased energy efficiency. Multiple works have attempted to solve this problem in various contexts [GBH12, MV15, LSM15]. The DAPHNE approach to exploiting the available heterogeneous resources at runtime, apart from selecting specific devices to execute entire operations, relies on the tiled execution engine that allows data parallelism within and across devices, taking into consideration the data locality, NUMA effects, and data transfer costs.

4.4.2 Node-level Scheduling (Local Scheduling)

NUMA-aware scheduling: one of the critical aspects that the **DAPHNE local scheduler** needs to consider is being aware of and minimizing the impact of the NUMA topology and NUMA aspects on application performance. A compute node is typically organized in a NUMA topology by assembling multiple (at least two) sockets in a single shared-memory system. The NUMA organization implies that the performance of a task, as defined in Section 1.1.3, may depend on 1) the architecture of compute node and 2) the task's code and its data locality characteristics. For instance, a task that operates on an input set that fits in the cache hierarchy is not expected to be affected by different NUMA placement choices, i. e., remote memory accesses occur very rarely. The NUMA effect may more severely affect a task that frequently experiences cache misses. Furthermore, NUMA topologies increase memory bandwidth, as they allow the cores of each NUMA socket to have direct access to separate local memory modules through separate memory controllers, whereas in uniform memory topologies all the cores of the system have direct access to the same memory modules through the same memory controllers. Hence, specific placement options for application tasks may improve their performance thanks to the increased memory bandwidth, even though specific memory references will take longer to serve, because they require fetching data from remote sockets.

The implications of NUMA topologies on scheduling are well known and many mechanisms have been proposed to alleviate them [DFF13, PSM15, KKM17, GNK20]. Current operating systems (e. g., Linux) and libraries (e. g., *numactl*) provide the opportunity to user-space programs to control the placement of tasks on NUMA resources by assigning them to specific cores/sockets and memory modules. The

DAPHNE local scheduler will leverage these mechanisms during execution and use them by incorporating sophisticated decision-making logic based on the characteristics of the tasks (i. e., code characteristics, execution behavior on the system based on performance counters, etc.). We will also consider cases where extending the underlying OS and library mechanisms would further enable the DAPHNE local scheduler to optimize performance, as well as other metrics such as throughput and resource utilization.

Resource sharing: the **DAPHNE local scheduler** also needs to be aware of and minimize the impact of resource interference in application performance. The processing cores of a compute node typically share some resources depending on the architecture of each chip, e. g., compute cores may share the L2, the Last-level Cache (LLC), or the memory controller for accessing a (local or remote) memory module. Hence, tasks executed on these cores experience resource interference. For instance, when the scheduler concurrently executes two different tasks on two cores that share the same LLC, this co-execution introduces interference if both tasks heavily use LLC. This interference may significantly decrease the application performance. Resource interference is also a well-known problem, and multiple prior works have proposed solutions to solve it [MTH11, TMS11, DK13, LCG16, NPG18]. Resource interference does not only arise on CPUs. Depending on the work sharing and co-execution capabilities of the underlying devices, it may manifest on GPUs, FPGAs, and computational storage setups. Hence, to fully leverage the computational capabilities of such systems that allow the co-execution of different tasks, the scheduling logic needs to be aware of the various sources of interference, avoid it, and mitigate it.
The DAPHNE local scheduler will consider the potential of **co-executing tasks** for suffering from and for introducing resource interference. We plan to enable the DAPHNE local scheduler to predict resource interference based on modeling components and mitigate it via certain performance monitoring components. We will also leverage code-level information as extracted by the various compilation steps and runtime characteristics based on the running behavior of the application tasks.

4.4.3 Cluster-level Scheduling (Distributed Scheduling)

The **DAPHNE distributed scheduler** is responsible for selecting the cluster's compute nodes that will be involved in computation and assigning them tasks. A significant amount of prior research has focused on scheduling distributed applications on clusters, with some works taking into consideration the application DAG [GCA16, GKR16] and even the heterogeneity within nodes [KK17], while others being agnostic of application DAGs [IPC09, GSG16, TZP16]. The DAPHNE approach is to leverage both the application DAG and the tiled execution engine to distribute work among nodes of a cluster. The DAPHNE distributed scheduler will consider locality, holding metadata information about prior executed tasks and the involved datasets and working sets to minimize the cost of data transfers, maximize parallelism, and improve performance.

5 Summary and Outlook

This deliverable describes the initial design of the DAPHNE scheduler for pipeline and tasks and the scheduling context, namely scheduling levels, decisions, and optimization goals. We explored the possible scheduling principles, strategies, and techniques that each component of the DAPHNE system (e. g., compiler and runtime) may consider. The DAPHNE local and distributed schedulers have been discussed with certain highlights on their different optimization goals.

In the next project period, we will extend the scheduler design for pipelines and tasks with hierarchical considerations. A **hierarchical approach** to scheduling of pipelines and tasks will require the exchange of scheduling information between the **local schedulers** (at the intra-node level) and the **global scheduler** (at the inter-node level). Hierarchical scheduling will aid in refining the scheduling decisions at a given level based on the available information about the current scheduling workload at other levels [EC21] [EC19]. For instance, monitoring the individual execution progress of each local scheduler and sharing such information with the global scheduler enables load balancing decisions across local schedulers [WZY+14]. Instantaneous reports about idle execution units allows a global scheduler to reuse them and to increase system utilization [EC21].

References

[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

[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.

[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)

[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)

[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)

[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)

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

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

[CAB+01] Anthony T. Chronopoulos, Razvan Andonie, Manuel Benche, and Daniel Grosu. A Class of Loop Self-scheduling for Heterogeneous Clusters. In Proceedings of International Conference on Cluster Computing,2001, pages 282-291.

[CB08] Ricolindo L. Cariño and Ioana Banicescu. Dynamic Load Balancing with Adaptive Factoring Methods in Scientific Applications. Journal of Supercomputing, 44(1):41-63, 2008.
[CG17] Chen, Quan, and Minyi Guo. Task scheduling for multi-core and parallel architectures. Singapore: Springer Nature, 2017.
[DFF13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13). Association for Computing Machinery, New York, NY, USA, 381–394. DOI: https://doi.org/10.1145/2451116.2451157.
[DK13] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13). Association for Computing Machinery, New York, NY, USA, 77–88. DOI: https://doi.org/10.1145/2451116.2451125.
[DP91] Chen, Ding-Kai, and Pen-Chung Yew. An empirical study on DOACROSS loops. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1991.
[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.
[EN16] R. Elmasri and S. B. Navathe. Fundamentals of Database Systems, Seventh Edition, Chapter 19. Pearson, 2016.
[GCA16] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, USA, 65–80.
[GKR16] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: packing and dependency-aware scheduling for data-parallel clusters. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, USA, 81–97.
[GNH+11] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, Yannis Sismanis: Large-scale matrix factorization with distributed stochastic gradient descent. KDD 2011: 69-77.
[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.
[EC20] Eleliemy, Ahmed, and Florina M. Ciorba. "A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems." Journal of Computational Science 51 (2021): 101284.
[EC21] A. Eleliemy and F. M. Ciorba. A Resourceful coordination Approach for Multilevel Scheduling. In Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2021), virtual event.
[FHSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. Journal of Communications of the ACM, 35(8):90-101, 1992.
[FHSU+96] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in Heterogeneous Systems via Weighted Factoring. In Proceedings of the 8th annual ACM symposium on Parallel algorithms and architectures, 1996, pages 318-328.
[GBH12] Gregg, Chris & Boyer, Michael & Hazelwood, Kim & Skadron, Kevin. 2012. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data.
[GNK20] David Gureya, João Neto, Reza Karimi, João Barreto, Pramod Bhatotia, Vivien Quema, Rodrigo Rodrigues, Paolo Romano, Vladimir Vlassov, "Bandwidth-Aware Page Placement in NUMA," in 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, 2020 pp. 546-556.
[GSG16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: fast, centralized cluster scheduling at scale. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, USA, 99–115.

[HHH+21] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner: Simplicity Done Right for Join Ordering. CIDR 2021.
[HS82] T. C. Hu, M. T. Shing: Computation of Matrix Chain Products. Part I. SIAM J. Comput. 11(2): 362-373 (1982).
[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).
[IPC09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). Association for Computing Machinery, New York, NY, USA, 261–276. DOI: https://doi.org/10.1145/1629575.1629601
[IPE+21] Ihde, Nina, Paula Marten, Ahmed Eleliemy, Gabrielle Poerwawinata, Pedro Silva, Ilin Tolovski, Florina M. Ciorba, and Tilmann Rabl. (2021) "A Survey of Big Data, High Performance Computing, and Machine Learning Benchmarks." In Proceedings of In Proceedings of the 13th Transaction Processing Council Technology Conference on Performance Evaluation & Benchmarking.
[KD19] K. Dursun et al. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. PVLDB, 12(12), 2019.
[KKM17] J. B. Kotra, S. Kim, K. Madduri and M. T. Kandemir, "Congestion-aware memory management on NUMA platforms: A VMware ESXi case study," 2017 IEEE International Symposium on Workload Characterization (IISWC), 2017, pp. 146-155, DOI: 10.1109/IISWC.2017.8167772.
[KK17] Michael Kaufmann and Kornilios Kourtis. 2017. The HCl Scheduler: Going all-in on Heterogeneity. In 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17). USENIX} Association.
[KW85] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. IEEE Transactions [PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Transactions on Computers, 100(12):1425-1439,1987.
[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
[MTH11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). Association for Computing Machinery, New York, NY, USA, 248–259. DOI: https://doi.org/10.1145/2155620.2155650.
[MV15] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Comput. Surv. 47, 4, Article 69 (July 2015), 35 pages. DOI: https://doi.org/10.1145/2788396.
[LCG16] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. ACM Trans. Comput. Syst. 34, 2, Article 6 (May 2016), 33 pages. DOI: https://doi.org/10.1145/2882783.
[Leu04] Leung, Joseph YT, ed. Handbook of scheduling: algorithms, models, and performance analysis. CRC press, 2004.
[LSM15] J. Lee, M. Samadi and S. Mahlke, "Orchestrating Multiple Data-Parallel Kernels on Multiple Devices," 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015, pp. 355 366, DOI: 10.1109/PACT.2015.14.
[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.
[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.
[N11] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4(9): 539-550 (2011).

[NPG19] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019). Association for Computing Machinery, New York, NY, USA, Article 15, 1–10. DOI: https://doi.org/10.1145/3337821.3337891.
[PD22] P. Damme, et. al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. 2022.
[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Transactions on Computers, 100(12):1425-1439, 1987.
[PSM15] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. Proc. VLDB Endow. 8, 12 (August 2015), 1442–1453. DOI: https://doi.org/10.14778/2824032.2824043.
[TMS11] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2011. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11). Association for Computing Machinery, New York, NY, USA, 12–21. DOI: https://doi.org/10.1145/2000417.2000419.
[TN93] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. IEEE Transactions on Parallel and Distributed Systems, 4(1):87-98, 1993.
[TZP16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16). Association for Computing Machinery, New York, NY, USA, Article 35, 1–16. DOI: https://doi.org/10.1145/2901318.2901355.
[SYT07] Wen-Chung Shih, Chao-Tung Yang, and Shian-Shyong Tseng. A Performance-based Parallel Loop Scheduling on Grid Environments. Journal of Supercomputing, 41(3):247-267, 2007.
[Ull75] J. D. Ullman. NP-complete Scheduling Problems. Journal of Computer and System Sciences, 10(3):384-393, 1975.
[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.
[VL14] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In SIGMOD, 2014.
[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
[WZY+14] Wang, Yizhuo, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. An adaptive and hierarchical task scheduling scheme for multi-core clusters. Parallel computing 40, no. 10 (2014): 611-627.