

D5.3 Improved Prototype of Pipeline, Task, and Parameter Server Scheduling



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

Version 1.0

PUBLIC



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

Document Description

This document describes the improved pipeline and task scheduling prototype implemented in the DAPHNE system. As discussed in previous deliverable reports (specifically, D4.1, D5.1, and D5.2), the DAPHNE project team has been consistently improving and expanding the scheduling mechanisms. The information presented in this document is based on a snapshot of the DAPHNE system prototype that implements well-tested and verified scheduling strategies.

The improved prototype of pipeline, task, and parameter server scheduling include usability improvements, the addition of automated scheduling to the DAPHNE scheduling framework (DAPHNEsched), the implementation of inter-node scheduling infrastructure, and parameter server updates.

D5.3 Improved Prototype of Pipeline, Task, and Parameter Server Scheduling			
WP5 – Scheduling and Resource Sharing			
Type of document	Report	Version	1.0
Dissemination level	PU	Project month	36
Lead partner	UNIBAS		
Author(s)	Jonas H. Müller Korndörfer (UNIBAS), Ahmed Eleliemy (UNIBAS up to 31/08/23), and Florina M. Ciorba (UNIBAS)		
Reviewer(s)	Pinar Tözün (ITU) and Marius Birkenbach (KAI)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Initial draft	Jonas H. M. Korndörfer
V0.2	Revised draft	Jonas H. M. Korndörfer / Florina M. Ciorba
V0.3	Added Section 4 Parameter Servers	Matthias Boehm / Jonas H. M. Korndörfer
V0.4	Integration of examples	Jonas H. M. Korndörfer
V0.5	Final draft	Jonas H. M. Korndörfer
V0.7	Improved text formatting	Jonas H. M. Korndörfer
V0.8	Review 1	Marius Birkenbach / Jonas H. M. Korndörfer
V0.9	Review 2	Pinar Tözün / Jonas H. M. Korndörfer
V1.0	Final version	Jonas H. M. Korndörfer / Florina M. Ciorba

Abbreviations

Abbreviation	Term
ITU	IT University of Copenhagen
KAI	Kompetenzzentrum Automobil- und Industrieelektronik
UNIBAS	University of Basel
WP	Work package

Terminology

Term	Definition
Snapshot	Since the DAPHNE system is under continuous development, a snapshot of the DAPHNE system refers to an immutable version of the project sources at a particular point in time.

Table of Contents

1	<i>Introduction</i>	4
2	<i>Usability Updates</i>	4
3	<i>Automated Scheduling</i>	6
3.1	<i>Automated Scheduling Usage</i>	7
4	<i>Inter-node Scheduling</i>	9
4.1	<i>Inter-node Scheduling Usage</i>	11
5	<i>Parameter Server</i>	12
6	<i>Conclusion and Outlook</i>	14
	<i>References</i>	15

1 Introduction

The scheduling philosophy within the DAPHNE system considers four key scheduling decisions. These decisions encompass work partitioning, work assignment, execution ordering, and execution timing. The scheduler design for pipelines and tasks within the DAPHNE system is documented in Deliverable 5.1 [D5.1]. The initial prototype is described in Deliverable 5.2 [D5.2] which documents multithreading scheduling implementation in DAPHNE, examples of how to use and extend the prototype, and all the initially available multithreading scheduling, work partitioning, and work assignment options.

This document describes the improved prototype of pipeline, task, and parameter server scheduling. The improved prototype consists of usability improvements (Section 2), automated scheduling implementation (Section 2), inter-node scheduling support (Section 3), and parameter server updates (Section 4).

2 Usability Updates

The usability updates implemented in the improved prototype consist of a series of changes that influence the way a user can access the different scheduling strategies in DAPHNE. We briefly describe below the steps required to install DAPHNE and provide an overview of the usability updates. We use the help menu of DAPHNE to highlight the changes from the initial to the improved prototype.

The DAPHNE system prototype is publicly accessible on GitHub, in the development repository at <https://github.com/daphne-eu/daphne>. **This document uses a snapshot that contains the newly introduced functionalities of the improved prototype.**

The installation steps below follow the same sequence as for [D5.2]. These steps are kept here for completeness. The usability updates are described after the installation instructions with examples regarding the changes.

Step 1 Download the snapshot from:

<https://daphne-eu.know-center.at/index.php/s/8XZz2ynDS6F3pNe>

```
unzip daphne-improved-sched-prototype-d5.3.zip
cd daphne
```

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.

```
This program compiles and executes a DaphneDSL script.
USAGE: daphne [options] script [arguments]
OPTIONS:
Advanced Scheduling Knobs:
--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 requi-
re profiling information as FSC
=PSS - Probabilistic self-scheduling
=AUTO - Automatic partitioning
--queue_layout=<value> - Choose queue setup scheme:
=CENTRALIZED - One queue (default)
=PERGROUP - One queue per CPU group
=PERCPU - One queue per CPU core
--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
--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)
--pin-workers - Pin workers to CPU cores
--pre-partition - Partition rows into the number of queues before applying scheduling technique
--vec - Enable vectorized execution engine
...
--dist_backend=<value> - Choose the options for the distribution backend:
...
=MPI - Use message passing interface for internode data exchange
...
Newly introduced support for distributed runtime scheduling using MPI, see Section 3
```

Four variables were added and are highlighted in the snippet above. Those are: “partitioning”, “queue_layout”, “victim_selection”, and “distributed_backend”. In the initial version of the prototype, the user would need to directly specify the expected configuration with the available options. For example, in the initial version of the prototype, to select the partitioning strategy, the user would need to directly use --STATIC when executing a DAPHNE script. To improve clarity and reduce the number of different options/variables in DAPHNE, we introduced the variables mentioned above which, depending on their value, define the different scheduling and partitioning configurations. Also highlighted in the snippet above is the automated scheduling and partitioning strategy namely AUTO. This is discussed in the following section.

3 Automated Scheduling

The choice of a scheduling algorithm and work partitioning strategy is not trivial. An inefficient choice can result in performance loss due to load imbalance and/or high scheduling overhead. Manually finding the highest-performing scheduling and work partitioning strategy is time-consuming and creates a large burden on the user. **To mitigate this issue, in the improved DAPHNE scheduling prototype, we introduce an additional automated scheduling and work partitioning strategy.**

The automated scheduling and work partitioning strategy implemented in the improved DAPHNE’s scheduling prototype was previously proposed and exhaustively evaluated in [MKEC’22]. In DAPHNE, such strategy is denoted as “AUTO” and can be used by defining the **-partitioning=<value> to AUTO**. When defining the “partitioning” strategy of DAPHNE to AUTO, the scheduler will automatically find a high performing task grain size.

The current implementation of AUTO consists of automatically defining a *grain size* for the scheduling and work partitioning strategy SS. The *grain size* works as a threshold, in the sense that when the size of a task, calculated by a scheduling and partitioning strategy, fall below the *grain size* threshold it will be replaced by the defined *grain size*. Declaring a proper *grain size* improves performance by reducing the number of scheduling rounds and decreasing loss of data locality. In a related work which uses OpenMP (a well-known and widely used parallel runtime for multithreaded applications), the authors have shown that SS with a proper *grain size* (referred to as *chunk parameter* in [KEMC’21]) frequently outperforms all other techniques [KEMC’21].

The automated *grain size* is calculated in DAPHNE based on the total number of possible tasks and the number of workers. The calculation uses the equation below (adapted from [MKEC’22]) to arrive at a high-performing *grain size* for SS. In Eq. 1, N denotes the total number of tasks, and P is the number of workers. The automated (AUTO) *grain size* is a practical method that uses the golden ratio $\phi = 1.618$ in the interval $[N/(2P), \dots, 1]$ to arrive at a *grain size* value that leads to high performance, using only N , P , and ϕ .

$$\text{Eq 1.) } \textit{grainSize} = \left\lfloor \frac{N}{2^f \times 2P} \right\rfloor, \text{ where } f = \left\lfloor \log_2 \left(\frac{N}{P} \right) \times \frac{1}{\phi} \right\rfloor$$

3.1 Automated Scheduling Usage

This section uses the snapshot discussed in Section 1.1 to exemplify the functionality of AUTO. **After downloading and installing DAPHNE from this snapshot, one can find the folder example-D5.3** inside the uncompressed DAPHNE folder (see installation steps in Section 1.1). This folder contains three files that will be required for the following example.

Files in /daphne/example-D.53:

- **components.daph** – connected components implementation in DAPHNE script used for the example. *This was selected to keep consistency between [5.2] and this deliverable.*
- **amazonx1.mtx** – input matrix for the components.daph DAPHNE script.
- **amazonx1.mtx.meta** – required metadata that characterizes **amazonx1.mtx** matrix.
- **matrix_addition.daph** – matrix summation DAPHNE script. This example is used later example in Section 3.

Step 1 Navigate to the main DAPHNE folder.

```
cd daphne
```

Step 2 Execute the *components.daph* DAPHNE script

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

```
./bin/daphne --vec --select-matrix-repr --timing --num-threads=1 --partitioning=AUTO ./example-D5.3/components.daph G=\"./example-D5.3/amazonx1.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": 0.00353753, "parsing_seconds": 0.0114381, "compilation_seconds": 0.0736943, "execution_seconds": 11.3002, "total_seconds": 11.3889}
```

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

```
./bin/daphne --vec --select-matrix-repr --timing --num-threads=6 --partitioning=AUTO ./example-D5.3/components.daph G=\"./example-D5.3/amazonx1.mtx\"
```

The output below shows the increased performance with more workers:

```
{"startup_seconds": 0.00386959, "parsing_seconds": 0.00983812, "compilation_seconds": 0.0680012, "execution_seconds": 5.18662, "total_seconds": 5.26833}
```

Figure 1 shows the results of a simple benchmark comparing AUTO and the other scheduling and work partitioning options available in DAPHNE. In this example, the SS scheduling and work partitioning strategy **without the AUTO grain size was not considered** as it causes too much scheduling overhead and loss of data locality. 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 1 are based on the execution of the following command repeatedly 5 times for each different “--partitioning” strategy.

```
# 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}
./example-D5.3/components.daph G="./example-D5.3/amazonx1.mtx"
```

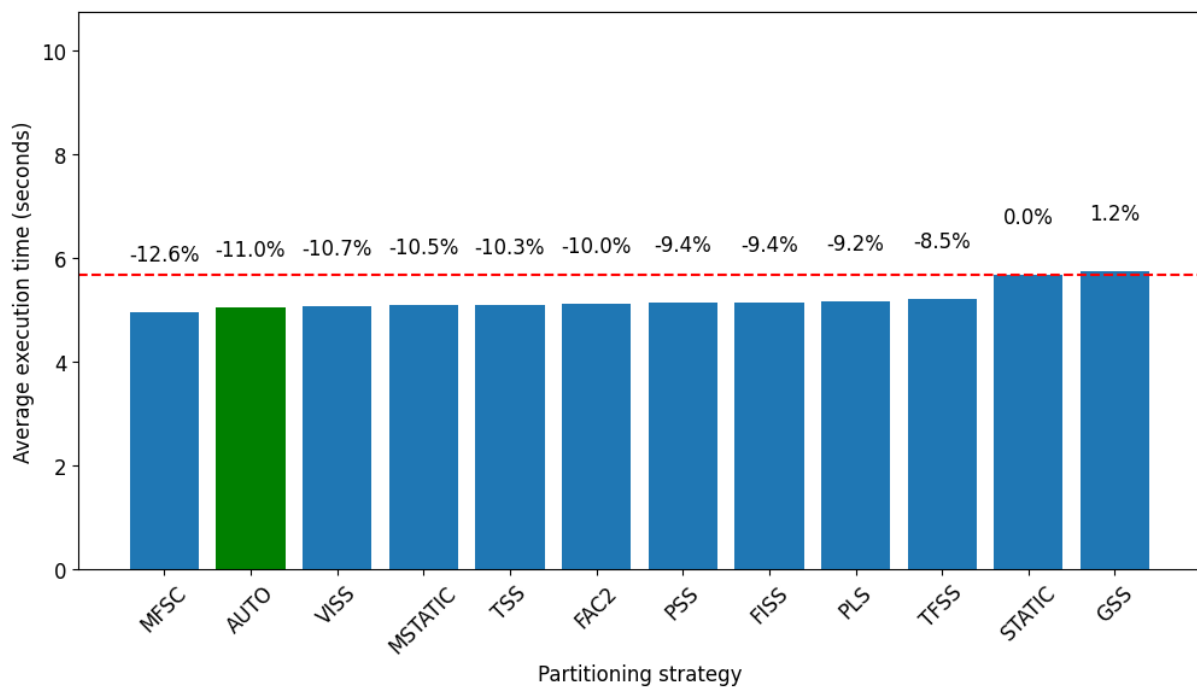


Figure 1. Connected Components DAPHNE script execution with different work partitioning strategies. The red dashed line highlights the performance of the script executing with STATIC (default).

In Figure 1, The x-axis shows the different scheduling and work partitioning options available in DAPHNE, while the y-axis shows the average execution time of the *components.daph* DAPHNE script. To improve visualization, the newly introduced AUTO work partitioning option is highlighted in green, while all other options are colored in blue. The X-axis is ordered by execution time, from the fastest to the slowest configuration. The dotted red line highlights the performance achieved by STATIC (default). The percentages highlight the execution time difference between a given work partitioning strategy and STATIC. Therefore, negative percentages represent faster executions.

We can observe that AUTO outperform most, but not all, other scheduling and work partitioning options. This shows that more development towards automated scheduling and work partitioning is required in DAPHNE. The development of automated selection of scheduling and work partitioning techniques during execution is still work in progress.

To reproduce the example shown in Figure 1, one can find two python scripts in the provided snapshot. (1) *D5.3-benchmark-example.py* that can be used to perform all executions,

and (2) *D5.3-benchmark-example-plotting.py* used to plot the results. The *D5.3-benchmark-example.py* script generates an output file called *output-benchmark-D5.3-example.txt* which contains all results and is required by the *D5.3-benchmark-example-plotting.py* script to plot the results. **Note that the Python scripts require the Matplotlib and Pandas packages to be installed.**

To use the scripts mentioned above, one can use the following steps:

Step 1 Navigate to the main DAPHNE folder

From there, execute the *D5.3-benchmark-example.py* script to perform the executions.

```
python3 D5.3-benchmark-example.py
```

While the above is executing, the script will print the progress of the execution which should look similar to the example below. Only after the script finishes, the complete output is stored in the *output-benchmark-D5.3-example.txt* file.

```
Iteration 1 of work partitioning strategy: STATIC
Command executed: ./bin/daphne --vec --select-matrix-repr --timing --num-threads=6 --partitioning=STATIC
./example-D5.3/components.daph G="./example-D5.3/amazonx1.mtx"
Partial output: {'startup_seconds': 0.00214975, 'parsing_seconds': 0.00533891, 'compilation_seconds':
0.0460168, 'execution_seconds': 5.94518, 'total_seconds': 5.99868, 'partitioning': 'STATIC'}
Iteration 2 of work partitioning strategy: STATIC
. . .
```

Step 2 Execute the *D5.3-benchmark-example-plotting.py* script

To plot the results, one can use the following command:

```
python3 D5.3-benchmark-example-plotting.py
```

The command above will generate and display the plot. Furthermore, it will store the plot in a file called *D5.3-benchmark-example-plot.png*.

4 Inter-node Scheduling

Message Passing Interface (MPI), is pivotal not only in High-Performance Computing (HPC) but also in diverse fields like data science. As the most widely adopted framework, MPI provides a standardized communication protocol, enabling parallel processes to exchange data effectively. In the computational landscape, where power arises from numerous interconnected nodes, MPI's importance is evident in its ability to seamlessly coordinate nodes, allowing concurrent execution.

The improved scheduling prototype of DAPHNE now implements support for inter-node scheduling with MPI. The initial design of the scheduler can be found in [D5.1], [EC'23], and [ECa'23]. Figure 2 shows the expansion of the DAPHNE scheduler design to accommodate distributed-memory systems.

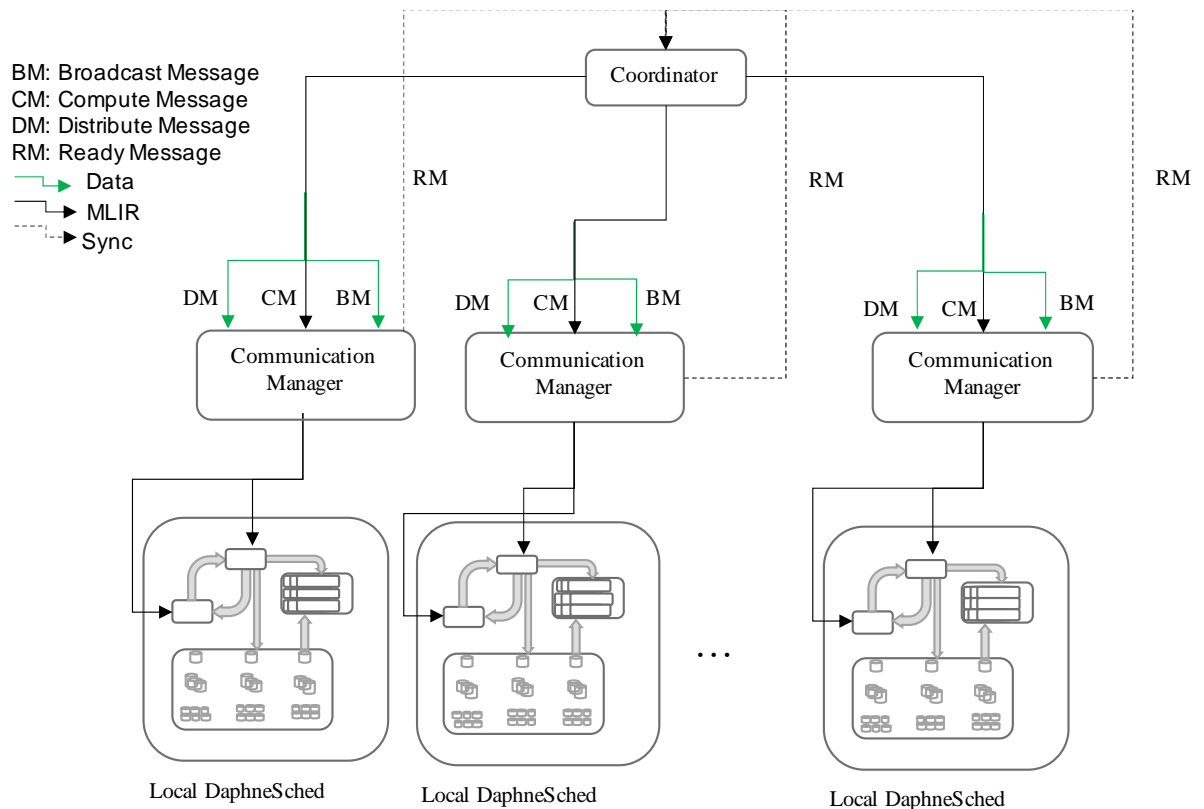


Figure 2. DAPHNE scheduler design for inter-node scheduling, taken from [ECa'23].

This design leverages the existing DAPHNE framework for shared-memory systems. The distributed version introduces a novel component referred to as the coordinator, which interfaces with multiple instances of the shared-memory schedulers. The coordinator acts as the primary entry point used by the DAPHNE runtime system. In other words, the DAPHNE runtime system communicates with the coordinator to manage the division, distribution, and collection of tasks and results from the various local scheduler instances. The modifications made to the local scheduler instances are minimal, primarily involving their ability to listen to incoming messages from the coordinator.

The coordinator is responsible for transmitting diverse messages and data. These include, for example, 1) distributed pipeline inputs, 2) broadcast pipeline inputs, and 3) MLIR code. On the receiving end, the worker component accepts and stores incoming data items as they arrive. Once a DAPHNE worker receives the MLIR code, it initiates the process of generating local tasks and executing them. Currently, the inter-node DAPHNE coordinator scheduler only supports STATIC scheduling and work partitioning, while the local schedulers support different dynamic strategies (see Section 2, and [D5.2]). Support for inter-node dynamic (and automated) scheduling and work partitioning is under development. Section 3.1 shows an example of how to execute a DAPHNE script using the inter-node scheduler and MPI.

4.1 Inter-node Scheduling Usage

To execute a DAPHNE script using the Inter-node scheduling implementation using MPI, one can use the following steps:

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 *matrix_addition.daph* is used. This is also available in the snapshot discussed in Section 1. The script can be found in the folder `/scripts/examples/`.

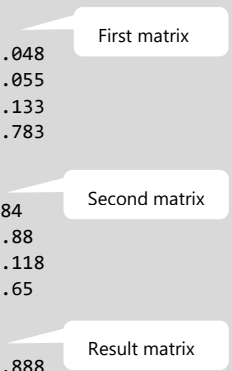
Step 2 Execute the *matrix_addition.daph* DAPHNE script

The following command executes the example with the distributed runtime:

```
mpirun -np 2 ./bin/daphne --timing -vec --num-threads=1 --distributed --dist_backend=MPI ./example-D5.3/matrix_addition.daph
```

The option `--distributed` activates the distributed runtime, and the option `--dist_backend=MPI` indicates that it should use MPI as backend. The output of the above command will be similar to the following:

```
part of random matrix m1
DenseMatrix(4x4, double)
198.775 920.847 442.355 743.048
601.304 666.923 172.892 546.055
979.946 88.0811 397.535 150.133
812.155 529.164 351.775 790.783
part of random matrix m2
DenseMatrix(4x4, double)
462.339 818.61 471.804 154.84
787.252 293.225 572.014 161.88
420.559 56.6451 95.3896 165.118
246.444 415.208 386.613 489.65
m3 = m1 + m2
DenseMatrix(4x4, double)
661.114 1739.46 914.158 897.888
1388.56 960.148 744.906 707.935
1400.5 144.726 492.925 315.25
1058.6 944.372 738.388 1280.43
{"startup_seconds": 0.0290332, "parsing_seconds": 0.00473754, "compilation_seconds": 0.0601155,
"execution_seconds": 14.7998, "total_seconds": 14.8937}
```



In the example above, only 1 thread per process (`--num-threads=1`) was used. In total, this execution used 2 workers (`mpirun -np 2`, 1 worker thread per process). It is also possible to combine multiple levels of parallelism and scheduling to increase performance. For example, it is possible to use a combination of number of processes and workers per process that fits the requirements of the workload. This example can scale up to approximately 6 workers.

The following command shows an example of how to combine parallelism and scheduling options at thread and MPI process level in DAPHNE:

```
mpirun -np 3 ./bin/daphne --timing --vec --num-threads=2 --partitioning=AUTO --distributed --dist_backend=MPI
./example-D5.3/matrix_addition.daph
```

With the command above, 3 MPI processes will be created, each with 2 threads that will execute the workload. This results in a total of 6 workers. Also, at thread level, the local DAPHNE scheduler will partition and schedule the local workload using the newly introduced AUTO strategy to achieve execution load balancing (see Section 2 for more information regarding AUTO). The output of the command above will be similar to the first example using the *matrix_addition_for_mpi.daph* DAPHNE script with 2 processes. The only difference is the performance improvement as one can see below:

```
. . .
{"startup_seconds": 0.0288519, "parsing_seconds": 0.00471287, "compilation_seconds": 0.0713904,
"execution_seconds": 6.84299, "total_seconds": 6.94794}
```

5 Parameter Server

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. 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].

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-compressed communication [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 enable researchers to explore combined configurations of such techniques for reducing communication overhead, the ETH team (around Ce Zhang who meanwhile left ETH Zurich and the DAPHNE project) created the standalone BAGUA system [G+21]. BAGUA provides well-defined communication alternatives at extension hooks and thus, allows experimenting with different 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>

It can be tested through the prepared tutorials at:

<https://tutorials.baguasys.com/>.

Integration into DAPHNE: We aim to eventually integrate the lessons learned and selected components from BAGUA into a parameter-server primitive in the DAPHNE system as well:

```
Mp = paramserv(M, X, y, updateGrad, updateModel, ASP, BATCH, 200, 64, ...)
```

Here, the parameter server is invoked through a built-in function to which we pass a list of initialized weight matrices M , the feature matrix X , the labels y , two function pointers for computing gradients and updating the model, as well as a synchronization strategy and frequency, number of epochs, batch size, and other parameters. While BAGUA focuses on variability (selecting and combining prepared communication alternatives), DAPHNE further aims for extensibility allowing the integration and experimentation with new communication and synchronization primitives. This DAPHNE integration will reuse existing functionality for second order functions (e.g., $\text{map}(X, \text{fun})$) but needs additional infrastructure for extensibility.

6 Conclusion and Outlook

This deliverable provides a comprehensive overview of the improved DAPHNE scheduling prototype. It describes key enhancements in usability, automated scheduling and work partitioning, and scheduling with distributed data developments. Automated scheduling solves the problem of relying on extensive experimentation or user expertise to select efficient scheduling algorithms. The developments regarding scheduling with distributed data enhance the scalability of DAPHNE. Finally, all these implementations together improve user experience and expand the scheduling capabilities of DAPHNE.

Further thread-pinning strategies and interfacing with resource managers are subject to improvement in future versions of the prototype. The automated grain size selection presented in this deliverable can be extended for the automated selection of scheduling and work partitioning strategies. This is part of ongoing work in WP5. Finally, support for dynamic and automated scheduling and work partitioning at MPI process level is an immediate next step. These forthcoming enhancements will provide users with even more adaptable and efficient scheduling solutions in DAPHNE.

References

- [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
- [D4.1] DAPHNE: D4.1 DSL Runtime Design, 11/2021.
- [D5.1] DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks, 11/2021.
- [D5.2] DAPHNE: D5.2 Prototype of Pipelines and Task Scheduling Mechanisms, 11/2022
- [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
- [EC'23] A. Eleliemy and F. M. Ciorba. "DaphneSched: A Scheduler for Integrated Data Analysis Pipelines", In Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Computing (ISPDC), Bucharest, Romania, 07/2023.
- [ECa'23] A. Eleliemy and F. M. Ciorba. "DaphneSched: A Scheduler for Integrated Data Analysis Pipelines", arXiv version <https://arxiv.org/abs/2308.01607>, 07/2023.
- [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
- [KEMC'21] J. H. Müller Korndörfer, A. Eleliemy, A. Mohammed, F. M. Ciorba. "LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications", in IEEE Transactions on Parallel and Distributed Systems (TPDS), 10/2021
- [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
- [MKEC'22] A. Mohammed, J. H. Müller Korndörfer, A. Eleliemy, F. M. Ciorba. "Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP", In IEEE Transactions on Parallel and Distributed Systems (TPDS), 12/2022
- [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, Shравan 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>
- [ZR14] Ce Zhang, Christopher Ré: DimmWitted: A Study of Main-Memory Statistical Analytics. Proc. VLDB Endow. 7(12): 1283-1294 (2014)