D5.2 Prototype of Pipelines and Task Scheduling Mechanisms

DAPHNE

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

Version 1.4

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 use of the pipeline and task scheduling mechanisms currently supported in the DAPHNE system. As discussed in previous deliverable reports [D4.1, D5.1], the DAPHNE project team continuously refines, extends, and adds scheduling mechanisms in DAPHNE. Nevertheless, extensibility is at the heart of the DAPHNE system. This offers DAPHNE users the opportunity to add scheduling mechanisms of their own. Therefore, this document describes how users can add or extend specific scheduling options.

All information presented in this document is based on a snapshot of the DAPHNE system prototype that implements well tested and verified scheduling strategies.

D5.2 Prototype of pipelines and task scheduling mechanisms			
WP5– Prototype of pipelines and task scheduling mechanisms			
Type of document	R	Version	1.4
Dissemination level	PU	Project month	24
Lead partner	UNIBAS		
Author(s)	Ahmed Eleliemy (UNIBAS) and Florina M. Ciorba (UNIBAS)		
Reviewer(s)	Pinar Tözün (ITU) and Marius Birkenbach (KAI)		
Contributors	all		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial draft	Ahmed Eleliemy
V1.1	Edits and clarity	Florina M. Ciorba
V1.2	Addresses comments from Florina M. Ciorba	Ahmed Eleliemy
V1.3	Clarity refinements	Ahmed Eleliemy, Florina M. Ciorba
V1.4	Final version	Ahmed Eleliemy, Florina M. Ciorba

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.

1 Scheduling Philosophy in DAPHNE

The scheduling philosophy in the DAPHNE system considers four types of scheduling decisions: *work partitioning*, *work assignment*, *execution ordering* and *execution timing*.

- *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 sizes).
- *Work assignment* is the mapping (or placing) the above units of work (or tasks) onto individual software units of execution (CPU threads). The DAPHNE compiler choses the type of hardware execution unit (CPU cores), while the DAPHNE runtime system currently schedules units of work across software execution units of the same type, e.g., CPU threads. For additional details, refer to Deliverables D5.1 and D4.1, sections 3.6 and 3.2.3, respectively.
- *Execution ordering* describes the order in which the tasks are to be executed. We rely on the DAPHNE vectorized execution engine, therefore, tasks within a vectorized pipeline have no dependencies and can be executed in arbitrary order.
- *Execution timing* denotes the times at which the tasks are to begin execution within the assigned software execution units (e.g., CPU threads).

Deliverable 5.1 [D5.1] describes the scheduler design for pipelines and tasks in the DAPHNE system. One may refer to Deliverable 5.1 for more details concerning these four types of scheduling decisions in the DAPHNE system. Currently, the DAPHNE system gives fine-grain control over the **first two scheduling decisions**, *work partitioning* and *work assignment*.

Work Partitioning: The current snapshot of the DAPHNE prototype implements twelve work partitioning schemes: *Static* (STATIC), *Self-scheduling* (SS), *Guided self-scheduling* (GSS), *Trapezoid self-scheduling* (TSS), *Trapezoid Factoring self-scheduling* (TFSS), *Fixed-increase self-scheduling* (FISS), *Variable-increase self-scheduling* (VISS), *Performance loop-based self-scheduling* (PLS), *Modified version of Static* (MSTATIC), *Modified version of fixed size chunk self-scheduling* (MFSC), and *Probabilistic self-scheduling* (PSS). These work partitioning schemes define the granularity of the tasks generated and scheduled by the DAPHNE system. These schemes divide tasks into chunks that range in size from N/P to 1, where N is the number of tasks and P is number of workers. For a detailed description of each partition scheme, see Section 4.1.1.1 in [D5.1]. This multitude of schemes existed in the literature to address application load imbalance profiles with minimum scheduling overheads [EC'21].

Work Assignment: The current snapshot of the DAPHNE prototype supports two work assignment mechanisms: *Single centralized work queue* (CENTRALIZED) and *Multiple work queues*.

When work is centralized in a single work queue, the workers (processes, CPU threads) follow the **self-scheduling principle**, which states that whenever a worker is free and idle, it obtains a task (or a group of tasks) from a central work queue.

When work is distributed across multiple work queues, the workers (CPU threads) follow the **work-stealing principle**, which states that whenever workers are free, idle, and have no other tasks in their local queue, they find a victim worker and steal tasks from the victim's work queue.

Distributed work queues can be *one per worker* (PERCPU) or *one per group of workers* (PERGROUP).

Work-stealing requires that workers employ a victim selection mechanism to find a victim overloaded worker and steal work from its queue. The currently supported victim selection mechanisms are: steal from the next adjacent worker (SEQ), steal from the next adjacent worker, but prioritize same NUMA domain (SEQPRI), steal from a random worker (RANDOM), and steal from a random worker, but prioritize same NUMA domain (RANDOMPRI).

The approach taken by DAPHNE regarding the latter **two scheduling decisions**: *execution ordering* and *execution timing* is as follows:

Execution ordering: In the initial design discussed in D5.1., the DAPHNE compiler takes care of the dependencies between individual operator pipelines. However, tasks within one pipeline have no dependencies. Thus, the DAPHNE runtime has full flexibility to execute them in any order, i.e., the software execution units can execute tasks in any order. However, since the implementation relies on First-in-first-out (FIFO) queue-based data structures to store partitioned work, tasks are executed following the first-come-first-serve (FCFS) principle. In this case, first-come means first partitioned, while first-serve means first executed. Other strategies will be supported in DAPHNE in the future, such as using task priorities set given their memory requirements.

Execution timing: Currently, whenever a worker obtains a task, it immediately begins executing it. In the future, we will explore other strategies where workers delay (defer) the execution of certain tasks to relieve the contention over specific resources, e.g., memory sub-systems.

2 Access to the DAPHNE System

The DAPHNE system prototype is publicly accessible on GitHub, in the development repository at https://github.com/daphne-eu/daphne. This document uses the latest release of DAPHNE, specifically release 0.1 (https://github.com/daphne-eu/daphne/releases/tag/0.1). This release corresponds to the commit with the following hash key: **aa6ef4a20254e12517c7a8acb5beee755d1726f3.**

Step 1 Get the DAPHNE source code as follows:

From https://daphne-eu.know-center.at/index.php/s/Zk9HbAZx343LPNADownload daphne-0.1.tar.gz

```
tar -xzf daphne-0.1.tar.gz
cd daphne
```

Step 2 Install dependencies: set up a Linux environment (tested with Ubuntu v. 20.04), and install the software dependencies 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.

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
```

One can skip this build step and directly go to Step 4 (execution) by using the precompiled DAPHNE release 0.1 https://github.com/daphne-eu/daphne/releases/download/0.1/daphne-0.1-bin.tgz.

We use the connected components algorithm to demonstrate all DAPHNE's scheduling knobs.. The amazon0601 co-purchase dataset (https://snap.stanford.edu/data/#amazon) is also used as input to the algorithm. We do not ship the DaphneDSL implementation of the connected components algorithm nor the Amazon co-purchase dataset with the precompiled release. Therefore, one needs to retrieve the following files

- Amazon0601_0.csv,
- Amazon0601_0.csv.meta, and
- components_read.daphne

available at https://daphne-eu.know-center.at/index.php/s/Zk9HbAZx343LPNA

Step 4 Execute the hello-world example with the following command:

```
./bin/daphne scripts/examples/hello-world.daph
```

The output of this command should look similar to the one below. The hello-world example generates two random dense matrices of size (2x3) and (3x2), multiplies, and displays the result matrix (2x2) (the last displayed dense matrix).

```
DenseMatrix(2x3, double)
161.297 117.379 135.027      First matrix
147.907 199.183 113.272
DenseMatrix(3x2, double)
161.297 147.907
117.379 199.183              Second matrix
135.027 113.272
DenseMatrix(2x2, double)
58026.7 62531.5
62531.5 74380.7              Result matrix
Hello world!
Bye!
```

Step 5 To learn about the scheduling options in DAPHNE, execute the following command:

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

The output of this command will show all DAPHNE's compilation and execution parameters including the **scheduling options**. The output below shows only the scheduling options that are presented in this deliverable. In the DAPHNE context, a task refers to the smallest unit of work to be scheduled, i.e., task partitioning and work partitioning schemes are synonymous (See Deliverable D5.1).

```
This program compiles and executes a DaphneDSL script.
USAGE: daphne [options] script [arguments]
OPTIONS:
Advanced Scheduling Knobs:
  Choose task partitioning scheme:
      --STATIC             - Static (default)
      --SS                 - Self-scheduling
      --GSS                - Guided self-scheduling
      --TSS                - Trapezoid self-scheduling
      --FAC2               - Factoring self-scheduling
      --TFSS               - Trapezoid Factoring self-scheduling
      --FISS               - Fixed-increase self-scheduling
      --VISS               - Variable-increase self-scheduling
      --PLS                - Performance loop-based self-scheduling
      --MSTATIC            - Modified version of Static, i.e., instead of n/p, it uses n/(4*p) where n is number
of tasks and p is number of threads
      --MFSC                - Modified version of fixed size chunk self-scheduling, i.e., MFSC does not require
profiling information as FSC
      --PSS                - Probabilistic self-scheduling
Choose queue setup scheme:
      --CENTRALIZED        - One queue (default)
      --PERGROUP           - One queue per CPU group
      --PERCPU             - One queue per CPU core
  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
…
```

3 Scheduling with DAPHNE

Deliverable D4.1 [D4.1], describes the DAPHNE system and the role of its vectorized (tile) execution engine to exploit parallelism within computing nodes. The vectorized execution engine decides the partitioning and assignment of work during applications' execution. Therefore, the option **`--vec`** is required by all scheduling options described in this document.

3.1 Multithreading Options

Number of threads: A DAPHNE user can control the total number of threads spawned by the DAPHNE runtime system by using the following parameter **`--num-threads`**. The default value of **`--num-threads`** is equal to the total number of physical cores of the host machine. This default value assumes the best performance is achieved when the number of threads is equal to the number of physical cores. This parameter takes a positive non-zero integer value. Illegal integer values, e.g., zero and negative values, will be ignored by the system and the default value will be used. Below is an example of how to use this option:

```
./bin/daphne --vec --num-threads=1 --select-matrix-representations --args f=\"./Amazon0601_0.csv\"
./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
2.27857
```

One may increase the total number of threads from 1 to 4 as follows:

```
./bin/daphne --vec --num-threads=4 --select-matrix-representations --args f=\"./Amazon0601_0.csv\"
./components_read.daphne
```

The output of the command should look similar to the following

```
Core algorithm time in seconds
1.14038
```

Thread Pinning: A DAPHNE user can decide if the DAPHNE system pins the parallel threads of the application to the underlying physical CPU cores. By default, the DAPHNE system does not pin its threads. Nevertheless, the DAPHNE system currently supports the round-robin pinning strategy, whereby threads are pinned to cores in increasing core ID order, with wraparound when the number of threads exceeds the available cores. Future pining strategies will consider cases when other applications execute on specific cores.

The option **`--pin-workers`** can be used to enable thread pinning in DAPHNE as follows:

```
./bin/daphne --vec --num-threads=4 --pin-workers --select-matrix-representations --args
f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
1.06243
```

Hyperthreading: If the –num-threads is not specified, the DAPHNE system sets the total number of application threads to the count of the **physical cores**. Nevertheless, if a host machine supports hyperthreading, a DAPHNE user can decide to use all available logical cores. When the user specifies **`--hyperthreading`**, the DAPHNE system sets the number of application threads to the count of the **logical cores (**which is typically double the count of physical cores)**.**

The output of the command should look similar to the following:

```
./bin/daphne --vec --hyperthreading --pin-workers --select-matrix-representations --args
f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

3.2 Work Partitioning Options

Partitioning scheme: A DAPHNE user selects the partitioning scheme by passing its name as an argument to the DAPHNE system. If the user does not specify a partitioning scheme, the default partitioning scheme (STATIC) will be used. This default value (STATIC) represents the straightforward parallelization and incurs the lowest scheduling overhead.
As an example, the GSS partitioning scheme is used as follows:

```
./bin/daphne --vec --num-threads=4 --pin-workers --GSS --select-matrix-representations --args
f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.854172
```

Task granularity: A DAPHNE user can exploit the **`--grain-size`** parameter to set the minimum size of the tasks generated by the DAPHNE system. The default value of **`--grain-size`** is 1, i.e., the data associated with a task represents 1 row of the input matrix. This parameter should be a positive non-zero integer value. Illegal integer values, e.g., zero and negative values, will be ignored by the system and the default value will be used.

The following command uses **`SS`** as a partitioning scheme with a minimum task size of 100 (rows):

```
./bin/daphne --vec --num-threads=4 --pin-workers --SS --grain-size=100 --select-matrix-representations --
args f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.666354
```

3.3 Work Assignment Options

Single centralized work queue: By default, the DAPHNE system uses a single centralized work queue. However, the user may explicitly use the **`--CENTRALIZED`** to ensure the use of single centralized work queue.

```
./bin/daphne --vec --num-threads=4 --pin-workers --SS --grain-size=100 --CENTRALIZED --select-matrix-
representations --args f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.677973
```

Multiple work queues: a DAPHNE user can exploit the use of multiple work queues by passing one of the following parameters **`--PERCPU`** or **`--PERGROUP`**. The two parameters cannot be used together, and if **`--CENTRALIZED`** is used with any of them, **`--CENTRALIZED`** will be ignored by the system. All queues exist within the same shared-memory system (one computing node). However, we plan to support distributing multiple work queue across distributed-memory nodes with Remote Procedure Call RPC and Message Passing Interface (MPI). We also plan to explore other work assignment options for distributed-memory systems.

- The **`--PERGROUP`** parameter ensures that the DAPHNE system creates as many groups as NUMA domains on the target host machine. The DAPHNE system assigns an equal number of workers (threads) to each group. Workers within the same group share a local work queue. The **`--PERGROUP`** parameter can be used as follows:

```
./bin/daphne --vec --num-threads=4 --pin-workers --GSS --PERGROUP --select-matrix-representations --args
f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.366549
```

- The **--PERCPU** parameter ensures that the DAPHNE system creates as many queues as workers (threads), such that each worker is assigned to a single local work queue. The **--PERCPU** parameter can be used as follows:

```
./bin/daphne --vec --num-threads=4 --pin-workers --GSS --PERCPU --select-matrix-representations --args
f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.469539
```

Victim Selection: A DAPHNE user can choose a victim selection strategy by passing one of the following parameters **--SEQ**, **--SEQPRI**, **--RANDOM,** and **--RANDOMPRI**. These parameters activate various victim selection strategies as follows

- **--SEQ** activates a sequential victim selection strategy, i.e., the ith worker steals form the (i+1)th worker. If the (i+1)th worker does not have work to be stolen, the ith worker tries to steal from the (i+2)nd worker, and so on. The last worker (ith = num-threads-1) steals from the first worker (ith=0). The Sequential victim selection strategy SEQ differs from the Round Robin strategy as follows: when a worker tries to steal from its successor neighbor, i.e., the ith worker can steal from only the (i+1)th worker as long as (i+1)th has work to be stolen [G'22].
- **--SEQPRI** is similar to **--SEQ** except that workers prioritize stealing from workers that are assigned to the same NUMA domain. When the host machine only has one NUMA domain, --SEQ and --SEQPRI work identically.
- **--RANDOM** activates a random victim selection strategy, i.e., the ith worker steals form a randomly chosen worker.
- **--RANDOMPRI** is similar to **--RANDOM** except that workers prioritize stealing from workers that are assigned to the same NUMA domain. When the host machine only has one NUMA domain, **--RANDOMPRI** and **--RANDOMPRI** work identically.
  In the absence of explicit values to either of these parameters, the DAPHNE system uses **--SEQ** as a default victim selection strategy.

As an example, the following command specifies **--SEQPRI** as a victim selection strategy:

```
./bin/daphne --vec --num-threads=4 --pin-workers --GSS --PERGROUP --SEQPRI --select-matrix-representations --
args f=\"./Amazon0601_0.csv\" ./components_read.daphne
```

The output of the command should look similar to the following:

```
Core algorithm time in seconds
0.404724
```

4 Extending Scheduling Strategies in DAPHNE

The use of a specific partitioning scheme has a significant influence on applications' performance. Indeed, partitioning schemes not only impact **the sizes of the generated tasks**, but they also indirectly impact the **work assignment** (following the self-scheduling or the work-stealing principle) and the granularity of the self-scheduled or stolen tasks, which influences the frequency of accessing the work queue(s). **Therefore, the current design ensures that DAPHNE developers can add custom work partitioning schemes.**

A DAPHNE developer can add a custom work partitioning scheme by changing the following files:

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

The LoadPartitioning.h file contains the implementation of the currently supported scheduling techniques (see D5.1 for details). The user should change two things:

1. The SelfSchedulingScheme enumeration. The user needs to add a name for the new technique, e.g., MYTECH:

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

2. The getNextChunk() function . This function uses a switch case to select the mathematical formula for determining the work partitioning that corresponds to the chosen scheduling method. The user needs to add a new case to handle the new technique.

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

The daphne.cpp file contains the code that parses the command line arguments and passes them to the DAPHNE compiler and runtime. The user needs to add the new technique as a valid parameter. Otherwise, the newly added technique cannot be used. The variable taskPartitioningScheme of type opt<SelfSchedulingScheme> should be extended with the declaration of **MYTECH** as follows:

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

The DAPHNE system needs to be rebuilt as shown earlier. Then, the new technique can be used as follows

```
./bin/daphne --vec --pin-workers --MYTECH --select-matrix-representations --args f=\"./Amazon0601_0.csv\"
./components_read.daphne
```

5 Conclusion and Outlook

This deliverable describes the scheduling strategies currently supported in the DAPHNE system. These scheduling strategies include task self-scheduling and work-stealing mainly for CPUs (For GPUs, FPGAs, and other accelerators, the scheduler follows the same task partitioning and queuing strategies [D7.1]). The deliverable also detailed the scheduling options that control the DAPHNE execution threads and their pinning. Information about extending specific scheduling options in the DAPHNE system is also provided.

Specific aspects, such as thread pinning strategies and interfacing with resource managers, are subject to improvement in upcoming deliverables.

For instance, the DAPHNE system works in execution environments managed by resource managers such as Slurm or YARN. Such resource managers often expose to user configuration options via environment variables. Currently, DAPHNE users have to pass the same configuration to the DAPHNE system, e.g., the number of threads and thread pining strategy. Future extensions will ensure that the DAPHNE system capture such information directly from the resource managers. We also plan to ultimately expose additional interfaces for greater extensibility, e.g., augmenting victim selection strategies and custom worker-to-work-queue mapping.

6 References

[D4.1] DAPHNE: D4.1 DSL Runtime Design, 11/2021.
[D5.1] DAPHNE: D5.1 Scheduler Design for Pipelines and Tasks, 11/2021.
[D7.1] DAPHNE: D7.1 Design of integration HW accelerators, 11/2021
[EC'21] A. Eleliemy and F. M. Ciorba, "A Distributed Chunk Calculation Approach for Self-scheduling of Parallel Applications on Distributed-memory Systems", In International Journal of Computational Science (JOCS)
[G'22] J. Giger, "Task Scheduling and Work Stealing in the DAPHNE Runtime System", Master's thesis, University of Basel, July 2022 (PDF).