D4.2 DSL Runtime Prototype

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

Version 1.3

PUBLIC

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

Document Description

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

In this deliverable we demonstrate the use of the DAPHNE runtime by sharing a snapshot of the DAPHNE prototype, and provide an example scenario of running simple DaphneDSL scripts on the local runtime as well as on the distributed runtime. We also demonstrate how to automate the deployment of the distributed DAPHNE system by using scripts. Finally, we provide information regarding the code organization, how the runtime system can be extended, and conclude with outlining some of its current limitations that will be addressed in future releases.

This demonstrator and document are the result of the collaborative work that is performed by all consortium partners that participate in WP4 "DSL Runtime and Integration", i.e., ICCS, KNOW, UNIBAS, ETH, ITU, HPI, and UM.

D4.2 DSL Runtime Prototype			
WP4 – DSL Runtime and Integration			
Type of document	R	Version	1.3
Dissemination level	PU	Project month	24
Lead partner	ICCS		
Author(s)	Aristotelis Vontzalidis (ICCS), Vasileios Karakostas (ICCS), Stratos Psomadakis (ICCS), Konstantinos Bitsakos (ICCS), Georgios Goumas (ICCS), Dimitrios Tsoumakos (ICCS), Florina M. Ciorba (UNIBAS), Ahmed Hamdy Mohamed Eleliemy (UNIBAS), Gabrielle Poerwawinata (UNIBAS)		
Reviewer(s)	Tilmann Rabl (HPI), Benjamin Steinwender (KAI)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial outline	All authors
V1.1	Updated content	Aristotelis Vontzalidis
V1.2	Updated content and polished text	All authors
V1.3	Update content based on comments from Tilmann Rabl and Benjamin Steinwender, and final polishing	Vasileios Karakostas, Aristotelis Vontzalidis, Dimitrios Tsoumakos

1 Background and Updates

To make this document self-contained, we briefly present an overview of DAPHNE and then provide a few important updates regarding the design and implementation of the DAPHNE distributed runtime system.

1.1 Background

DAPHNE works in a hierarchical approach [D4.1], by logically splitting the nodes of a cluster into the coordinator node and the compute worker nodes (Figure 1.1). The coordinator node implements the distributed runtime logic and is responsible for orchestrating the data and work distribution among workers. The coordinator sends necessary data and code to each worker by using its distribution primitives and receives their respective results. Each worker executes the received code through the local runtime system via the vectorized execution logic.

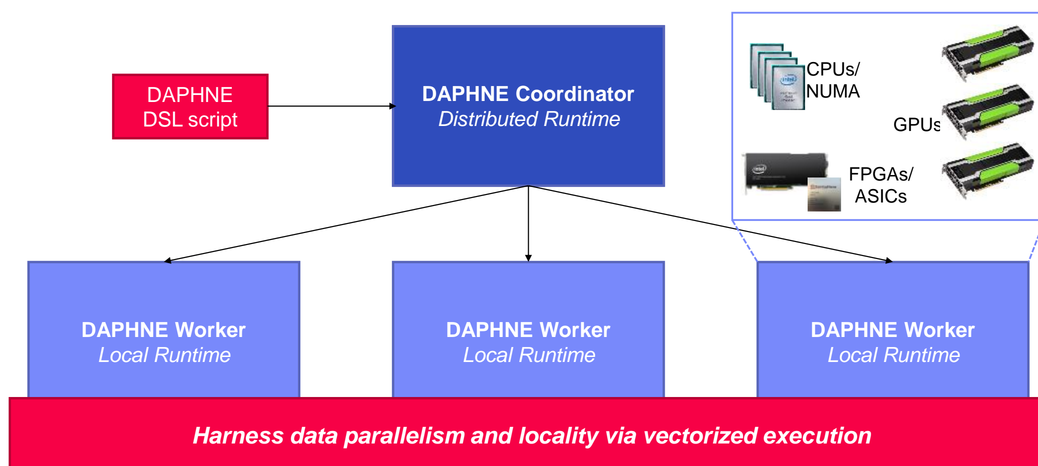

Figure 1.1: DAPHNE Hierarchical Approach.

The DAPHNE vectorized execution works by fusing multiple operations together and exploiting data parallelism [D+22]. Code changes are not required; instead, DAPHNE creates pipelines containing two or more operations. Data is split across multiple processing units (CPUs) and each processing unit works on a chunk of data. Figure 1.2 shows how the DAPHNE vectorized execution works locally at node level.

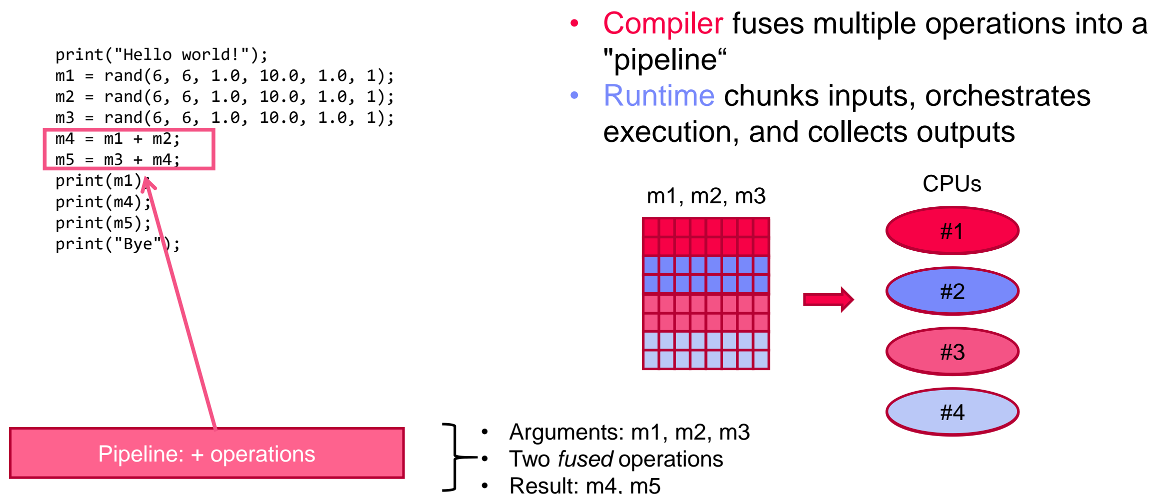

Figure 1.2: DAPHNE Local Execution.

1.2 Design and Implementation Updates

Deliverable D4.1 described the initial design of the DAPHNE distributed runtime system, where fused operator pipelines were not supported. During the second year of the project, the initial design and implementation were significantly refactored in order to provide such support. Instead of executing single "distributable" operations one after the other in distributed fashion, we can now fuse multiple operators into a "distributed pipeline" which is then computed on multiple workers. At runtime, the coordinator sends data chunks to the workers using the distribution primitives, and then the coordinator sends to the workers the MLIR code fragment of the pipeline (i.e., one or more fused operators) to be executed. Each worker locally compiles the received IR code fragment in order to optimize the code generation targeting its available resources (CPUs, GPUS, accelerators, etc.), and executes the generated code through the local runtime system via the vectorized execution engine.

The updated distributed runtime supports the vectorized execution engine and works very similarly to the local runtime (Figure 1.3). Instead of CPUs, there are multiple distributed nodes that receive chunks of data and perform computations on them. At the moment, the distributed-pipeline generation is heavily dependent on the vectorized engine that uses the same compiler and runtime techniques to fuse multiple operations together as in local execution.

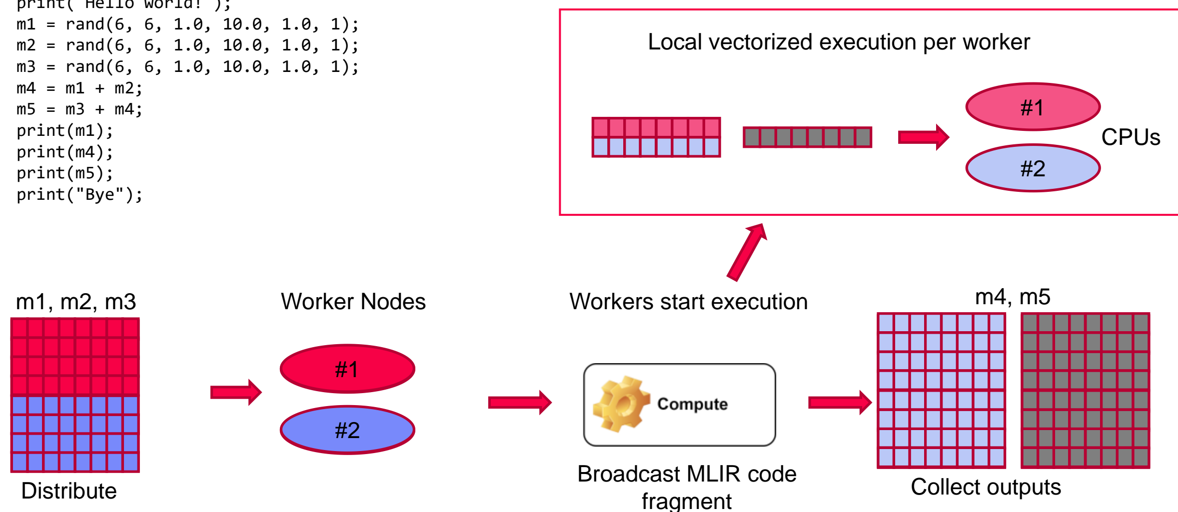

Figure 1.3: DAPHNE Distributed Execution.

Moreover, we decoupled the distributed runtime execution from the gRPC communication framework which was tightly linked in the past. This makes it easy to extend DAPHNE and integrate it with any communication framework. In that context, we integrated DAPHNE with the MPI library and we now provide primitive support for running the distributed DAPHNE system with MPI, in order to leverage its characteristics for improved performance.

Furthermore, we extended the I/O support for using multiple data formats, such as Arrow and Matrix market, in addition to the CSV format. Finally, we defined and implemented our own DAPHNE file format along with custom (de)serialization support to enable more efficient IO and network communication.

2 Artifact Access

The DAPHNE DSL runtime prototype that is described in this deliverable is publicly accessible as a snapshot of the DAPHNE development repository (created in November 7th, 2022) under the following link:

Link: https://daphne-eu.know-center.at/index.php/s/7biwzCQdYcSzgrF (22 MB)

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

3 Demonstration scenario

We present two demonstration scenarios. The first scenario targets the execution of DAPHNE on a single node using the local runtime, while the second scenario targets the execution of DAPHNE on a cluster of nodes using the distributed runtime. We use the following `example.daph` script to demonstrate the fusion of operations for both the local and the distributed runtime:

```
print("Hello world!");
m1 = rand(6, 6, 1.0, 10.0, 1.0, 1);
m2 = rand(6, 6, 1.0, 10.0, 1.0, 1);
m3 = rand(6, 6, 1.0, 10.0, 1.0, 1);
m4 = m1 + m2;
```

```
m5 = m3 + m4;
print(m1);
print(m4);
print(m5);
print("Bye");
```

3.1 Prerequisities

Step 1 Install Dependencies: Setup a Linux environment (tested with Ubuntu 20.04), and install the dependency versions specified in `docs/GettingStarted.md`, which includes clang, cmake, git, lld, ninja, pkg-config, python3, openjdk, gfortran, and uuid-dev. For tests related to DAPHNE's Python API, installing numpy in the Python environment is needed.

Step 2 Download and Extract: Download the artifact from the link in Section 2, and extract it as follows into a directory called daphne.

```
$ tar -xvf daphne.tar.gz
$ cd daphne
```

Step 3 Build DAPHNE: Build the DAPHNE prototype from inside the directory you created at step 2.

```
$ ./build.sh
```

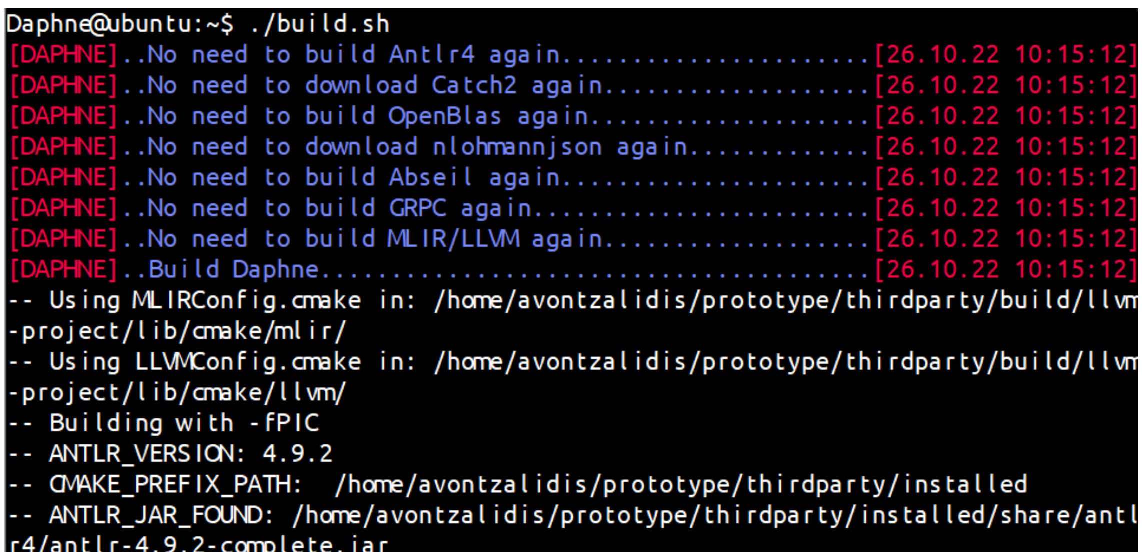

Figure 3.1: Building DAPHNE.

3.2 Local Runtime

Step 4 Run DAPHNE: Run the aforementioned example script on the local runtime:

```
$ ./bin/daphne example.daph
```

```
Daphne@ubuntu:~$ ./bin/daphne ./example.daph
Hello world!
DenseMatrix(6x6, double)
9.97466 9.39302 2.15312 9.99136 3.1248 4.56923
4.4912 7.02771 9.41985 8.6168 3.81946 5.72093
4.99108 3.06619 5.80973 9.22566 5.11484 4.87629
9.45215 8.0055 7.44373 8.22482 1.83521 5.66337
8.78518 8.46232 8.46643 3.45745 1.53319 7.03475
6.33759 7.04489 4.70609 2.77796 3.60667 2.27908
DenseMatrix(6x6, double)
19.9493 18.786 4.30624 19.9827 6.2496 9.13845
8.98239 14.0554 18.8397 17.2336 7.63892 11.4419
9.98215 6.13239 11.6195 18.4513 10.2297 9.75257
18.9043 16.011 14.8875 16.4496 3.67041 11.3267
17.5704 16.9246 16.9329 6.9149 3.06638 14.0695
12.6752 14.0898 9.41218 5.55592 7.21333 4.55816
DenseMatrix(6x6, double)
29.924 28.179 6.45936 29.9741 9.3744 13.7077
13.4736 21.0831 28.2596 25.8504 11.4584 17.1628
14.9732 9.19858 17.4292 27.677 15.3445 14.6289
28.3565 24.0165 22.3312 24.6745 5.50562 16.9901
26.3555 25.387 25.3993 10.3723 4.59957 21.1043
19.0128 21.1347 14.1183 8.33387 10.82 6.83724
Bye
```

Figure 3.2: Running the DAPHNE local runtime with an example script.

3.2.1 Vectorized engine

Step 5 Run DAPHNE with vectorized engine: Run the same script with the --vec flag:

```
$ ./bin/daphne --vec example.daph
```

Step 6 Explain DaphneIR: Run the same scenario with the additional explain flag in order to investigate the generated DaphneIR and observe the generated pipeline:

```
$ ./bin/daphne --vec --explain=kernels example.daph
```

3.3 Distributed Runtime

From the user perspective, utilizing the DAPHNE distributed runtime does not require any changes to the DaphneDSL script. From the system perspective, the distributed runtime requires communication support between the coordinator and the worker nodes. We currently provide support for using two different communication frameworks between the coordinator and the workers, i.e., gRPC (more mature-level support) and MPI (initial-level support).

3.3.1 gRPC

Distributed runtime with gRPC requires manually building and starting/deploying distributed workers, which communicate with the coordinator using the gRPC framework.

Step 4 Build DistributedWorker: DAPHNE's DistributedWorker can be built with:

```
$ ./build.sh --target DistributedWorker
```

```
Daphne@ubuntu:~$ ./build.sh --target DistributedWorker
[DAPHNE]..No need to build Antlr4 again.......................................[25.10.22 16:08:40]
[DAPHNE]..No need to download Catch2 again....................................[25.10.22 16:08:40]
[DAPHNE]..No need to build OpenBlas again.....................................[25.10.22 16:08:40]
[DAPHNE]..No need to download nlohmannjson again..............................[25.10.22 16:08:40]
[DAPHNE]..No need to build Abseil again.......................................[25.10.22 16:08:40]
[DAPHNE]..No need to build GRPC again.........................................[25.10.22 16:08:40]
[DAPHNE]..No need to build MLIR/LLVM again....................................[25.10.22 16:08:40]
[DAPHNE]..Build Daphne........................................................[25.10.22 16:08:40]
-- Using MLIRConfig.cmake in: /home/avontzalidis/prototype/thirdparty/build/llvm-project/lib/cmake/mlir/
-- Using LLVMConfig.cmake in: /home/avontzalidis/prototype/thirdparty/build/llvm-project/lib/cmake/llvm/
-- Building with -fPIC
-- ANTLR_VERSION: 4.9.2
-- CMAKE_PREFIX_PATH:  /home/avontzalidis/prototype/thirdparty/installed
-- ANTLR_JAR_FOUND: /home/avontzalidis/prototype/thirdparty/installed/share/antlr4/antlr-4.9.2-complete.
jar
-- ANTLR4_JAR_LOCATION is /home/avontzalidis/prototype/thirdparty/installed/share/antlr4/antlr-4.9.2-com
plete.jar
-- Antlr4 DaphneDSLGrammar - Building Common Include-, Source- and Tokenfiles, Visitor Include- and Sour
cefiles
-- Antlr4 SQLGrammar - Building Common Include-, Source- and Tokenfiles, Visitor Include- and Sourcefile
```

Figure 3.3: Building DAPHNE's distributed workers.

Step 5 Start Workers: After the DistributedWorker is built, manually execute as many workers as preferred and specify an IP and Port to listen to for each one. In this scenario two workers listen to `localhost` and use ports 50001 and 50002.

```
$ ./bin/DistributedWorker IP:PORT
```

```
Daphne@ubuntu:~ $./bin/DistributedWorker localhost:50001
Started Distributed Worker on `localhost:50001`
```

```
Daphne@ubuntu:~ $./bin/DistributedWorker localhost:50002
Started Distributed Worker on `localhost:50002`
```

Figure 3.4: Starting DAPHNE's distributed workers.

Step 6 Set environmental variables: The coordinator needs to know at which IPs and ports the workers listen on. For now, we use an environment variable "`DISTRIBUTED_WORKERS`" to specify a comma-separated list of workers addresses.

```
$ export DISTRIBUTED_WORKERS=localhost:50001,localhost:50002
```

Step 7 Run DAPHNE: Finally, after distributed workers are deployed and the environmental variable is set, DAPHNE can be executed in a distributed fashion by providing the `--distributed` switch.

```
$ ./bin/daphne --distributed example.daph
```

```
Daphne@ubuntu:~$ ./bin/daphne --distributed ./example.daph
Hello world!
DenseMatrix(6x6, double)
9.97466 9.39302 2.15312 9.99136 3.1248 4.56923
4.4912 7.02771 9.41985 8.6168 3.81946 5.72093
4.99108 3.06619 5.80973 9.22566 5.11484 4.87629
9.45215 8.0055 7.44373 8.22482 1.83521 5.66337
8.78518 8.46232 8.46643 3.45745 1.53319 7.03475
6.33759 7.04489 4.70609 2.77796 3.60667 2.27908
DenseMatrix(6x6, double)
19.9493 18.786 4.30624 19.9827 6.2496 9.13845
8.98239 14.0554 18.8397 17.2336 7.63892 11.4419
9.98215 6.13239 11.6195 18.4513 10.2297 9.75257
18.9043 16.011 14.8875 16.4496 3.67041 11.3267
17.5704 16.9246 16.9329 6.9149 3.06638 14.0695
12.6752 14.0898 9.41218 5.55592 7.21333 4.55816
DenseMatrix(6x6, double)
29.924 28.179 6.45936 29.9741 9.3744 13.7077
13.4736 21.0831 28.2596 25.8504 11.4584 17.1628
14.9732 9.19858 17.4292 27.677 15.3445 14.6289
28.3565 24.0165 22.3312 24.6745 5.50562 16.9901
26.3555 25.387 25.3993 10.3723 4.59957 21.1043
19.0128 21.1347 14.1183 8.33387 10.82 6.83724
Bye
```

```
Daphne@ubuntu:~$ ./bin/DistributedWorker localhost:50000
Started Distributed Worker on `localhost:50000`
```

```
Daphne@ubuntu:~$ ./bin/DistributedWorker localhost:50001
Started Distributed Worker on `localhost:50001`
```

Figure 3.5: Running the DAPHNE distributed runtime with an example script.

3.3.2 MPI

The current MPI implementation is under heavy development and is not yet included in the main repository of DAPHNE or in the deliverable snapshot. However, we showcase what is currently implemented and provide an example of future additions.

The biggest difference between MPI and gRPC is that the former does not require the user to manually start and deploy workers. The MPI runtime handles all that, as long as the user specifies that MPI will be used as the distributed back-end. The distributed execution with MPI is much simpler and more straightforward for the user; however, it requires MPI to be pre-installed in the system. Note that the `--dist_backend` flag is not currently supported but will be added in a future version of DAPHNE.

Example use with 4 workers:

```
$ mpirun -n 4 ./bin/daphne --distributed --dist_backend=MPI example.daph
```

```
Daphne@ubuntu:~$ mpirun -n 4 ./bin/daphne --distributed --dist_backend=MPI ./example.daph
Hello world!
DenseMatrix(6x6, double)
9.97466 9.39302 2.15312 9.99136 3.1248 4.56923
4.4912 7.02771 9.41985 8.6168 3.81946 5.72093
4.99108 3.06619 5.80973 9.22566 5.11484 4.87629
9.45215 8.0055 7.44373 8.22482 1.83521 5.66337
8.78518 8.46232 8.46643 3.45745 1.53319 7.03475
6.33759 7.04489 4.70609 2.77796 3.60667 2.27908
DenseMatrix(6x6, double)
4.65278e-310 0 4.65278e-310 4.65269e-310 4.65278e-310 1.2732e-313
0 4.65278e-310 4.65278e-310 4.65278e-310 4.65278e-310 6.24993e-321
14.9732 9.19858 17.4292 27.677 15.3445 14.6289
28.3565 24.0165 22.3312 24.6745 5.50562 16.9901
17.5704 16.9246 16.9329 6.9149 3.06638 14.0695
12.6752 14.0898 9.41218 5.55592 7.21333 4.55816
DenseMatrix(6x6, double)
29.924 28.179 6.45936 29.9741 9.3744 13.7077
13.4736 21.0831 28.2596 25.8504 11.4584 17.1628
4.44659e-323 4.65278e-310 1.2732e-313 0 0 0
0 0 0 4.44659e-323 4.65278e-310 1.2732e-313
26.3555 25.387 25.3993 10.3723 4.59957 21.1043
19.0128 21.1347 14.1183 8.33387 10.82 6.83724
Bye
```

Figure 3.6: Running the MPI DAPHNE distributed runtime with an example script.

3.4 Deployment of distributed runtime

`deploy/deployDistributed.sh`

This script automates the task of building, packaging, deploying, and managing distributed workers on remote machines. Note that this script is intended to be used along with the gRPC distributed back-end, since MPI does not require any prior deployment.

Prerequisites: Before using this script, SSH configuration must be set within it.

Running `deployDistributed.sh`: The script is invoked from the root DAPHNE directory.

```
$ ./deploy/deployDistributed.sh --help
```

```
#************************
# SSH Configurations
#************************

identity_file=
USERNAME=
sshPort=22
```

Figure 3.7: SSH configuration within the deployDistributed.sh script

```
Daphne@ubuntu:~$ ./deployDistributed.sh --help
Start the DAPHNE distributed worker on remote machines.
Usage: ./deployDistributed.sh [-h|--help] [--deploy] [--pathToBuild] [-r| --run]
 [-s| --status] [--kill] [-peers IP[:PORT], ...]

Please remember to set DISTRIBUTED_WORKERS=IP:PORT,IP:PORT,... before running a
DAPHNE script.
Logs can be found at [pathToBuild]/logs

You can specify [ip:port, ...] list inside the script or pass it as argument.
Default port is 50000 but if you are running in local
machine you need to specify ports [-peers IP[:PORT], ...]

--deploy:
This includes downloading and building all required third-party
material (if necessary), for building the DistributedWorker.
You should only invoke it from DAPHNE's root directory
(where this script resides).

Optional arguments:
  -p, --port              Specify port number (default 50000).
  -i identity_file        Specify identity file (default: default ssh private ke
y).
  --deploy                Compress and deploy build folder to remote machines.
  --pathToBuild           A path to deploy or where the build is already deploye
d (default ~/DaphneDistributedWorker can be specified in the script).
  --peers [IP[:PORT],...] Specify (comma delimited) IP:PORT workers (default loc
alhost:50051,localhost:50052)
  -r, --run               Run workers on remote machines.
  -s, --status            Get distributed workers' status.
  --kill                  Kill all distributed workers.
  -h, --help              Print this help message and exit.
Daphne@ubuntu:~$
```

Figure 3.8: Usage of the deployDistributed.sh script.

Build and deploy the DAPHNE DistributedWorker at a specified path on remote machines:

```
$ ./deploy/deployDistributed.sh --deploy --pathToBuild
~/DaphneDistributedWorker --peers localhost:50051,localhost:50052
```

$ ./deploy/deployDistributed.sh --deploy --pathToBuild ~/DaphneDistributedWorker --peers localhost:50000,localhost:50001

Start, get status and terminate the DAPHNE DistributedWorkers:

```
$ ./deploy/deployDistributed.sh --run
$ ./deploy/deployDistributed.sh –status
$ ./deploy/deployDistributed.sh --kill
```

```
Daphne@ubuntu:~$ ./deploy/deployDistributed.sh --run
Starting remote workers at  localhost:50051
Starting remote workers at  localhost:50052
Daphne@ubuntu:~$ ./deploy/deployDistributed.sh --status
Checking remote worker localhost:50051
UID          PID    PPID  C STIME TTY      STAT   TIME CMD
avontza+    9538       1  0 18:33 ?        Sl     0:00 ./bin/DistributedWorker localhost:50051
Worker is UP
Checking remote worker localhost:50052
UID          PID    PPID  C STIME TTY      STAT   TIME CMD
avontza+    9537       1  0 18:33 ?        Sl     0:00 ./bin/DistributedWorker localhost:50052
Worker is UP
Daphne@ubuntu:~$ ./deploy/deployDistributed.sh --kill
Killing remote workers at  localhost
Killing remote workers at  localhost
```

Figure 3.9: Running and terminating the DAPHNE distributed workers.

4 Prototype Structure

We now present in detail how the runtime-related code of the DAPHNE prototype is organized, along with a brief description. The runtime source code is located at **`src/runtime/`**:

- **`distributed`** (distributed runtime source code)
  - **`coordinator`** (distributed coordinator code)
  - **`proto`** (gRPC and protobuf)
  - **`worker`** (distributed worker)
- **`local`** (local runtime source code)
  - **`context`** (runtime context information)
  - **`datagen`** (data generation utils)
  - **`datastructures`** (implementation of DAPHNE's data structures)
  - **`io`** (input/output)
  - **`kernels`** (implementation of DAPHNE's build in functions)
  - **`vectorized`** (vectorized engine)

5 Extending the DAPHNE Runtime

DAPHNE is designed to be highly extensible. A DAPHNE developer can easily implement additional DAPHNE built-in functions (kernels) for the local runtime. See more in the GitHub documentation section:

https://github.com/daphne-eu/daphne/blob/main/doc/development/ImplementBuiltinKernel.md

In addition, the distributed runtime is decoupled from the communication framework, making it easy to extend and implement additional options. See more about developing the distributed runtime here:

https://github.com/daphne-eu/daphne/blob/main/doc/development/ExtendingDistributedRuntime.md

6 Limitations

The distributed runtime is still under heavy development and currently there are various limitations. Most of these limitations will be fixed in future releases.

- The distributed runtime system currently depends heavily on the vectorized engine of DAPHNE and how pipelines are created and multiple operations are fused together. This causes some limitations related to pipeline creation (e.g., not supporting pipelines with different result outputs or pipelines with no outputs).
- The distributed runtime system currently supports only `DenseMatrix` types and `double` value types – `DenseMatrix<double>`.

```
Daphne@ubuntu:~$ ./bin/daphne --select-matrix-repr --distributed components.daphne n=60 e=30
Execution error: Distribute grpc only supports DenseMatrix<double> for now
```

- A DAPHNE pipeline input might exist multiple times in the input array. This is currently not supported. In the future, similar pipelines will simply omit multiple pipeline inputs and each one will be provided only once.

```
Daphne@ubuntu:~$ ./bin/daphne --distributed components.daphne n=60 e=30
Execution error: Distributed runtime only supports unique inputs for now (no duplicate inputs in a pipeline)
```

- Memory release at worker (node) level is not implemented yet. This means that after some time, the workers can could have their memory filled up completely, requiring a restart.

7 References

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

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

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

[D+22] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pınar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz Wrosz, Aleš Zamuda, Ce Zhang, and Xiao Xiang Zhu. "DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines", In 12th Annual Conference on Innovative Data Systems Research (CIDR 2022).