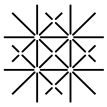

University of Basel

Benchmarking the DAPHNE System Infrastructure for Integrated Data Analysis Pipelines

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC Group
https://hpc.dmi.unibas.ch/

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor: Dr. Ahmed Hamdy Mohamed Eleliemy

Reto Krummenacher
reto.krummenacher@unibas.ch
03-054-327

July 14, 2022

Acknowledgments

I would like to express my deep gratitude to my supervisor Dr. Ahmed Hamdy Mohamed Eleliemy, for his continuous support and patient guidance in helping me navigate this interesting but challenging project. I would also like to thank my advisor Prof. Dr. Florina M. Ciorba, for providing constructive and helpful feedback and allowing me to complete my bachelor thesis as part of the HPC research group. My appreciation is also extended to Mr. Jonathan Giger, who was simultaneously doing his master thesis, for providing vital elements to work with DAPHNE successfully. Finally, I wish to thank Dr.-Ing. Patrick Damme from the DAPHNE project for the time he spent to help me understand DaphneDSL.

Abstract

Modern data analysis involves several stages, from big data management through high-performance computing to machine learning training. Combined, those form an integrated data analysis (IDA) pipeline. Although systems of those areas share many runtime techniques, the software stacks differ considerably. As a result, various programs, languages, and data representations are used in each data analysis step. DAPHNE is an open system infrastructure that provides all necessary tools for such IDA pipelines. So far, it has not been possible to evaluate DAPHNE with existing benchmarking applications, as it employs a domain-specific language (DSL) named DaphneDSL. This thesis surveys existing benchmark suits for implementable benchmarks and rewrites them accordingly. The process reveals several functionality limitations of DaphneDSL, aggravating or rendering impossible an implementation. Performance evaluation of the rewritten application against their reference is not conclusive and questions the comparability in general.

Table of Contents

1 Introduction

Data analysis has become one of the essential elements of modern research. The process involves several stages, from big data management through high-performance computing to machine learning training. Combined, these steps form an integrated data analysis (IDA) pipeline. Systems of this area share many compilation and runtime techniques [10]. Nevertheless, the software stacks differ considerably, resulting in various programs, languages, programming paradigms, and data formats used in each process step. The DAPHNE project aims to offer an open system infrastructure to provide all necessary tools for an IDA pipeline. In March 2022, the first prototype [40] was released. Even though some performance experiments were done for DAPHNE [10], a component that is missing so far is benchmarking. A benchmark is a standardized problem that serves as a basis for evaluating an infrastructure. The metrics collected during this assessment compare the systems under test against each other. There exist numerous known benchmark suites with many heterogeneous test applications. However, those are unusable as the DAPHNE system employs a domain-specific language (DSL) named DaphneDSL. The purpose of this thesis is to provide the means to benchmark DAPHNE. The questions to be answered: to what extent is it possible to rewrite existing benchmarks with DaphneDSL, and how are the implementations performing against their reference application? The process of finding a response is fourfold. First, find applications from existing benchmark suites to be implemented with DaphneDSL. Second, identify the advantages and limitations of the DAPHNE software infrastructure. Third, rewrite the selected benchmarks with DaphneDSL; fourth, evaluate the new implementations against their originals by designing factorial experiments. The thesis is structured as follows. Section 2 presents the collected information about existing benchmark applications, followed by an overview of the DAPHNE system and DaphneDSL in section 3. Section 4 elaborates on the benchmark-to-rewrite selection process, explains revealed issues, and describes the implementations. Finally, section 5 covers the experiments carried out to compare the DaphneDSL implementation with the reference benchmarks and discusses the obtained results.

2
Background

A benchmark is "a standardized problem or test that serves as a basis for evaluation or comparison" [30]. It also refers to the process of obtaining quantitative measures for a meaningful comparison across multiple systems [23]. Many benchmarks exist targeting big data (BD), high-performance computing (HPC), or machine learning (ML) systems [23]. In this work, ten are considered to some extend. HiBench [24], BigDataBench [3] and BigBench [16] are examples of BD benchmarks. HPC Challenge [19], HPCG [17], HPL-AI [21], UEABS [42] and SPEC [35] belong to the group of HPC benchmarks, whereas DeepBench [2] and MLPerf [28] are ML benchmarks.

All suites are listed in Table 2.1 together with their main target domain, the covered topics and the number of individual benchmarks offered. The remaining part of this chapter briefly describes each suite.

2.1 BD Benchmarks

2.1.1 HiBench

HiBench [24] is a BD benchmarking suite offering 29 workloads from 6 categories: Micro, Machine Learning, SQL, Web search, Graph, and Streaming. Micro benchmarks test a single functionality. In the case of HiBench, methods like sorting or word count are assessed. Those are making use of strings rather than numbers as input data. The other fields covered by HiBench are testing more complex tasks. Among others, classification, regression, and clustering are included in the ML benchmarks, by far the largest category. All implementations use third-party software (e.g., Hadoop, Spark). In particular, data generation and workload handling rely on methods provided by these additional software packages

2.1.2 BigDataBench

The current version 5.0 [3] is a unified and scalable BD and AI benchmark suite [15]. The main idea is to consider BD and AI workloads as a combination of different computation units called "data motifs": Matrix, Sampling, Logic, Transform, Set, Graph, Sort, and Statistic Computation. In contrast to the traditional methodology of creating new bench-

Table 2.1: Summary of the 10 benchmark suites, including main domain, the covered topics and the number of benchmarks from each topic.

Suite	Main domain	Covered topic	Number of benchmarks
HiBench	BD	Micro	6
		Machine Learning	13
		SQL	3
		Websearch	2
		Graph	1
		Streaming	4
BigDataBench V5.0	BD	Micro	27
		Component	16
		Application	2
BigBench	BD	Product retailer	1
HPC Challenge	HPC	Basic operations	6
HPCG	HPC	Basic operations	1
HPL-AI	HPC	Solving linear system	1
UEABS	HPC	Existing applications	13
SPEC	HPC	Cloud	1
		CPU	1
		Graphics and Workstation	6
		HPC: OpenMP, MPI	4
		Java Client/Server	4
		Storage	1
		Power	1
		Virtualization	2
DeepBench	ML	Basic operations	5
MLPerf Training	ML	Vision	4
		Language	2
		Commerce	1
		Research	1

marks for every possible workload, BigDataBench uses combinations of the eight data motifs to represent BD and AI workloads.

The suite includes micro benchmarks, each covering a single data motif, component benchmarks consisting of data motif combinations, and end-to-end application benchmarks. The latter is made of combined component benchmarks. Furthermore, BigDataBench contains real-world data sets like Wikipedia entries or Amazon movie reviews together with the big data generator suite (BDGS) used to generate the input for the benchmarks.

The micro benchmarks [6] are using different workloads and are available for several

software stacks like Hadoop, Spark, TenserFlow, or Hive. While the implementations working with Hadoop or Spark call the corresponding software's desired methods directly through shells, Sort, Grep, WordCount, MD5, RandSample, and FFT are using the Message Passing Interface (MPI) framework and are written in C++. In contrast to the first five, generating data with the BDGS, FFT has its own data generator written in C.

Similar to the micro implementations, the provided component benchmarks are based on different software stacks [5]. Among those using MPI are naive Bayes, K-means, or breadth-first search. Similar to FFT, K-means has its own data generator written in C++. The scripts used to run K-means are written in C. As metrics, the I/O and computation time are reported. For detailed instruction on how to run all the benchmarks together with an overview, the reader is referred to [4].

2.1.3 BigBench

BigBench is a big end-to-end data benchmark proposal [16] covering the variety, velocity, and volume aspects of data from a product retailer. The workload is designed around a set of queries covering different categories of BD analytics proposed by McKinsey. BigBench combines structured data like sales, semi-structured data (e.g., web logs), and unstructured data like reviews. The original proposal is the basis of TPCx-BB [41]. The Transaction Processing Performance Council (TPC) is a non-profit corporation founded in 1988 focusing on providing data-centric benchmark standards to the industry. TPCx-BB executes analytical queries in the context of retailers, measuring BD systems' performance. The source code is available under a license agreement only.[1]

2.2 HPC Benchmarks

2.2.1 HPC Challenge

HPC Challenge [26], as the name indicates, an HPC domain benchmark, is designed to measure a range of memory access patterns. The source code [20] is freely available. HPC Challenge suite consists of several tests, including:

- HPL - the High Performance Linpack benchmark solves a system of linear equations measuring the floating point rate.
- DGEMM - double precision real matrix multiplication, measuring the floating point rate.
- PTRANS - testing the communications between pairs of processors using a parallel matrix transpose exercise.
- FFT - measures the floating point rate of a complex one-dimensional Discrete Fourier Transformation (DTF).

[1] https://www.tpc.org/tpc_documents_current_versions/download_programs/tools-download-request5.asp?bm_type=TPCX-BB&bm_vers=1.5.2&mode=CURRENT-ONLY

Moreover, the suite contains a relatively unique test in the sense of the stressed component. STREAM measures the sustainable memory bandwidth (Bytes/s) for simple vector kernels copy, scale, add, and a combination of the last two called triad. The idea originates in the fact that CPUs have become much faster than the computer memory system, resulting in the memory bandwidth being the bottleneck of the system rather than the computational performance [29]. The source code is written in C and uses MPI.

Released originally by DARPA (Defense Advanced Research Projects Agency) High Productivity Computing Systems, HPC Challenge helps define performance boundaries of computing systems and tests the performance of HPC architectures.

2.2.2 HPCG

The high-performance conjugate-gradient benchmark [17] is intended to represent today's applications performance better than HPL [13]. HPL solves systems of linear equations employing Gaussian elimination, heavily relying on dense matrix multiplications showing stride memory access. In contrast, scientific computations governed by partial differential equations (PDE) tend to exhibit irregular memory access patterns [13]. For this reason, the computation to data-access ratio is very low for PDE algorithms compared to matrix multiplications where computation dominates rather than data access.

To account for the difference between the applications, HPCG applies a multigrid preconditioned conjugate-gradient algorithm to solve a three-dimensional PDE model problem represented by a sparse linear system. The equation at each point in the three-dimensional domain depends on the values of 26 surrounding neighbors [13], showing the importance of memory bandwidth rather than computing power. Besides basic operations like sparse matrix-vector multiplications and dot products, the algorithm uses Gauss-Seidel preconditioning for the conjugate-gradient solver on each multigrid level. The source code of the reference implementation [18] is freely available and written in C++.

2.2.3 HPL-AI

HPL-AI [21] seeks to highlight the convergence of HPC and AI workloads. Traditional algorithms for modeling phenomena in physics or biology require 64-bit accuracy. In contrast, ML applications desire 32-bit or even lower floating-point precision formats. This lesser demand delivers higher performance levels and energy savings.

HPL-AI is solving a linear system with LU decomposition in 32-bit precision (float). The results then are transformed back to 64-bit precision (double). The reference implementation [22] is written in C and is freely accessible.

2.2.4 UEABS

Another representative of the HPC domain is the Unified European Applications Benchmark Suite [42], a collection of 13 application codes with the objective to be run on Tier-1 (up to 1'000 cores) and Tire-0 (up to 10'000 cores) sized systems. Each has its own workload of several sizes to match the system under test. All applications are existing software packages

used in various fields, like physics (e.g., ALYA) or chemistry (e.g., CP2K). One example is named TensorFlow, after the open-source machine learning platform. This benchmark provides three test cases of different-sized networks, all using DeepGalaxy[2] which trains a deep neural network to classify galaxy merges in the universe. It is written in Python and built with TensorFlow. The dataset used is freely available.

2.2.5 SPEC

The Standard Performance Evaluation Corporation offers tools to evaluate the performance and energy efficiency of computer systems [35]. Currently, there are 20 different benchmarks from eight topics available: HPC, cloud, CPU, graphics and workstation performance, storage, power as well as Java performance. The four benchmarks dedicated to HPC are SPEC ACCEL, SPEChpc 2021, SPEC OMP 2012, and SPEC MPI 2007. As it is a commercial tool, the source code is not freely available.

2.3 ML Benchmarks

2.3.1 DeepBench

DeepBench [2] is an ML benchmark suite designed to test four basic operations crucial for deep learning. This includes general matrix multiply (GEMM), convolutions, recurrent layers, and all-reduce. DeepBench provides source code to benchmark those operations for hardware suppliers like Intel, NVIDIA, ARM, or AMD. While the NVIDIA examples are written in CUDA, the others are implemented in C++.

In fact, DeepBench uses vendor-supplied libraries. For instance, the Intel GEMM benchmark relies on Intels Math Kernel Library[3] (MKL) to perform matrix multiplications. Alongside the benchmark for dense matrices, one for sparse matrices is available.

2.3.2 MLPerf

As the name indicates, MLPerf is a machine learning benchmark suite that started in 2018. Nowadays, it is part of the MLCommons association [32], which provides training [28] and inference [34] benchmarks. Here, only the first is discussed in detail, focusing on the training stage of ML models, whereas the latter considers the inference stage, meaning the usage of a model on live data to produce output.

The MLPerf training benchmark addresses the unique challenges of benchmarking deep learning (DL) training [28]. For example, there is a trade-off between quality and performance optimization during the training process, which is not observable until the end of an entire training session. Another source of variation is the minibatch size. Larger minibatches can lower training time on distributed systems. However, they adversely affect learning dynamics and might need more epochs to reach the same accuracy. A certain

[2] https://github.com/maxwelltsai/DeepGalaxy
[3] https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-fortran/top/blas-and-sparse-blas-routines/blas-routines/blas-level-3-routines/gemm.html

stochastic influence like random weight initialization in DL training is also present, resulting in run-to-run variation for the same model with identical hyperparameters. Finally, multiple software frameworks for ML have emerged, distinctly executing similar computations. All these factors increase benchmarking complexity of DL training.

MLPerf v2.0 offers eight training benchmarks from four topics: vision, language, research and commerce [32]. The performance metric is the time-to-train necessary to reach a defined quality target. This includes auxiliary operations needed for training such models but neglects overhead like system initialization time. The source code for the reference benchmarks is available [33], and workloads are provided.

2.4 Convergence

The brief description of each benchmark suite shows the heterogeneity of the field. Benchmarks range from testing a single calculation to covering the full range of ML tasks, including data handling and training. The suites are grouped according to their main domain. However, an underlying tendency exists to cover not only one but two domains with a single benchmark. Hence, a convergence of domains [23].

An example is HiBench and the contained ML benchmarks. Despite providing ML purposes, the metrics used do not reflect ML performance [23]. Similar to HiBench are the component benchmarks of BigDataBench. Alongside BD character, they have an ML training aspect. Another interesting case with convergence is HPL-AI, with its underlying idea to do calculations at single precision before converting the result to double precision.

Based on the collected information, the next step is to decide which application is implemented in DaphneDSL. Before that, a brief introduction to the DAPHNE system is given in the next section.

3
Related Work

DAPHNE, denoting "integrated **D**ata **A**nalysis **P**ipelines for large-scale data management, **H**igh-performance computing, and machi**NE** learning" [37], aims to be an open and extensible system infrastructure for developing and executing IDA pipelines [10]. The prototype [40] has been publicly accessible since March 2022. The DAPHNE system architecture is shown in Figure 3.1. It is built from scratch in C++ utilizing MLIR, a multi-level intermediate representation (IR) aiming to tackle the challenges in programming language design when developing domain-specific IRs [25]. In addition, DAPHNE uses existing runtime libraries such as the basic linear algebra subprogram kernels [9].

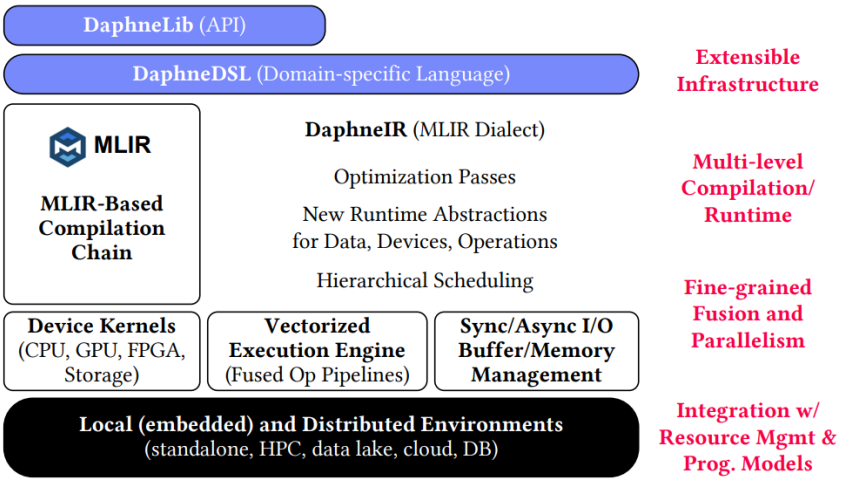

Figure 3.1: The DAPHNE system infrastructure. [10]

DAPHNE supports parallelism over hardware devices with the built-in vectorized execution engine [10]. Operations on a matrix are compiled into operator pipelines. The operator pipeline forms a vectorized task with its input data, the output, and a combine method. The inputs are neighboring row partitions of the matrix. Vectorized execution then appends tasks to queues executed by CPU workers, with a task being the scheduling unit. An illustration of the process can be found in [10], Figure 3. Not yet supported are parallel loop constructs [10].

Users may interact with DAPHNE via the Python API DaphneLIB or through a domain-specific language (DSL) similar to Python NumPy, or R. DaphneDSL [38] supports built-in operations and functions conditional flow control and abstract data types. A text file with a DSL program (e.g., script.daphne) is parsed into DaphneIR using ANTLR4 [1], for which a DSL grammar file and built-in methods are available [11]. Both were important sources of information because, at the time of writing this report, the DaphneDSL code documentation was only finished to a certain degree [12]. .

4
Methodology

The first objective is to find benchmarks to be implemented in DaphneDSL. Three exclusion and three selection criteria are defined to choose from the benchmark variety covered in section 2. Based on those, the benchmarks to rewrite are determined in a two-stage process. This section elaborates on those criteria before describing the two stages. Subsequently, current DaphneDSL limitations are presented. The last part of this section is dedicated to the implementation as well as details about the selected benchmarks.

4.1 Criteria

One of the most important requirements when considering the implementability of a certain benchmark is detailed knowledge about the performed operations. In order to comprehend the program parts of a benchmark, it is necessary to have access to the source code. Another important aspect is the variety of topics addressed by benchmarks. The fact that DAPHNE is still an ongoing project leads to a focus on testing simple tasks rather than full applications. More precisely, benchmarks not related to basic operation or functionality are discarded. A further exclusion argument, the reference implementation is building on third-party software like Hadoop or TensorFlow, seeing that those are not available in DaphneDSL.

Regarding the selection of benchmarks, it is desirable to implement those having some convergence, thus stressing two domains instead of one. Another criterion is based on the fact that DAPHNE aims to support IDA pipelines. Given the importance of simple matrix operations in many algorithms, the performance of such functionality in DAPHNE is of high interest. In other words, they are chosen first. At last, to ensure some variety, benchmarks testing not covered features are favored.

To summarize, the benchmark evaluation criteria are:

1. Exclusive: The Source code is not freely available without license agreement.
2. Exclusive: The benchmark is not testing basic operations or functionality.
3. Exclusive: The benchmark is building on third-party software.
4. Selecting: The benchmark combines two or more benchmarking domains.

5. Selecting: Favoring benchmarks testing operations common in linear algebra.

6. Selecting: The benchmark covers a feature not considered so far.

4.2 First Stage: Excluding Benchmarks

The exclusion stage starts with ruling out those violating the first criteria. This holds for TPC BigBench and SPEC. Consequently, the benchmarks provided by any of those suites are not considered for implementation.

The next step is to focus on those testing a basic operation or functionality, thus applying criterion two. As mentioned, micro benchmarks are meant to test some basic functionality. As shown in Table 2.1, UEABS and MLPerf Training measure performance with applications or rather complex ML tasks and are therefore excluded. In addition, all but the micro benchmarks from HiBench and BigDataBench are not taken into account because they are not dedicated to a basic operation.

Many suites use a third-party software, notably those with a certain ML purpose, such as MLPerf, BigDataBench, and HiBench. However, those were already excluded from further consideration as not testing basic operations. Consequently, this criterion is primarily satisfied. One exception is the micro benchmarks from HiBench and BigDataBench. HiBench, for instance, is using Hadoop or Spark. Since part of the Hadoop source code was untraceable, the micro benchmarks from HiBench are not considered any further.

In the case of BigDataBench, owing to the dependence on TensorFlow or Hive, all but those using MPI are neglected. In addition, the Connected Component benchmark is left out as it relies on the parallel boost graph library[4]. Lastly, the DeepBench suite is discarded as well. Even though testing simple operations like GEMM, the benchmarks are built on vendor-supplied libraries, thus violating the not-third-party criterion.

The result of the first process stage is a list with benchmarks displayed in Table 4.1 together with some information about the language and the total number of lines of code. It is worthwhile noting that this is a summary, with the LOC being the sum of all necessary scripts. The extended list, including references to the source code, can be found in Table A.1.

4.3 Second Stage: Selecting Benchmarks

By applying criteria four to six, the benchmarks to rewrite are selected. Regarding the fourth, HPL-AI is the only remaining benchmark with convergence and hence is the first to be rewritten. Speaking of criterion five, STREAM tests scalar multiplication as well as matrix additions while PTRANS tests parallel matrix transposing, both building blocks for many algorithms. Even though having comparatively high numbers of lines of codes, those two benchmarks are chosen too. Considering criterion six, the decision was in a fervor of Sort. In contrast to other benchmarks, Sort is working on strings. Testing this DaphneDSL

[4] https://github.com/BenchCouncil/BigDataBench_V5.0_BigData_MicroBenchmark/tree/main/MPI/MPI_Connect/parallel-bgl-0.7.0

Table 4.1: List of benchmarks after exclusion stage.

Suite	Benchmark	Domain	Language	LOC
HPL-AI	Solve Linear System	HPC, ML	C	744
BigDataBench	Sort	BD	C++	856
BigDataBench	Grep	BD	C++	488
BigDataBench	WordCount	BD	C++	513
BigDataBench	MD5	BD	C++	614
BigDataBench	RandSample	BD	C++	523
BigDataBench	FFT	BD	C, C++	209
HPC Challenge	DGEMM	HPC	C	157
HPC Challenge	STREAM	HPC	C	714
HPC Challenge	PTRANS	HPC	C	1612
HPC Challenge	FFT	HPC	C	1259
HPC Challenge	RandomAccess	HPC	C	1861
HPC Challenge	HPL	HPC	C	1923
HPCG	conjugate-gradient	HPC	C	1470

Note: LOC is the sum of lines of code. See Table A.1 for the LOC count breakdown.

feature is not covered so far. Interestingly, Fast Fourier Transformation (FFT) is provided by multiple suites and is selected likewise.

Additionally, a benchmark is included, which is not part of the list. Computing the Mandelbrot set is a frequently used performance test within the High Performance Computing group of the University of Basel. Implementing this has the further advantage of testing the handling of complex numbers in DaphneDSL.

Altogether, the benchmarks considered for implementation in DaphneDSL are HPL-AI, STREAM, PTRANS, FFT, Sort, and Mandelbrot. A hitherto ignored aspect is the available functionality in DaphneDSL itself. Given that DAPHNE is an ongoing project, limitations and possible flaws must be considered.

4.4 Issues with DaphneDSL

The benchmark implementation revealed several issues, which are categorized into two groups. The first includes commands that should work but did not (i.e., bugs). The second covers missing features, meaning functionality that would simplify working with DaphneDSL but is not available yet (i.e, nice to have). For all cases, issues have been opened on the GitHub repository of the DAPHNE prototype [40]. All opened issues are listed in Table A.2 together with their category and the current status. A minimal working example to reproduce each issue can be found in section C.1.1. Section C.1.2 provides code fragments with ideas and suggestions on how a nice to have functionality could be implemented. This report only covers newly opened issues related to this thesis. Not surprisingly, there are many more for an ongoing project like DAPHNE. In fact, when writing this report, there is

a total of 119 open issues.

Among the missing features is a function to return the object type. In particular, when working with casting, it is helpful to determine the type. Hence, a `typeof()` function would simplify coding and debugging. Code C.15 illustrates how this could work in DaphneDSL based on know R functionality. Another example is that of matrix literals. In the early stages of this project, DaphneDSL did not support setting up matrices with predefined non-sequential values. This was a tremendous limitation, notably when working with linear systems during the implementation of HPL-AI, as a given matrix had to be combined element-wise (Code C.13). This is supported by now and can be used as shown in Code C.14. Even though the non-availability of certain methods had increased difficulty, bugs had much more significant implications.

On the one hand, some flaws could be resolved in little time by the DAPHNE developer team, like the 'rbind row check' bug displayed in Code C.1 where `rbind()` was wrongly verifying the number of rows instead of the number of columns when combining two matrices. Additionally, certain bugs could be circumvented, like the parsing of $n - 1$ (Code C.2) or the proper indexing in loops (Code C.5).

On the other hand, certain issues had a direct effect on benchmark implementability. Above all, the 'shape change in loops' issue is shown in Code C.9. The bug prevents changing matrices inside a loop. Hence, modifying an element of a matrix with variable indices is not feasible in DaphneDSL. Furthermore, combining a changed entry with the unchanged array is currently not supported. In consequence, implementing benchmarks based on iterative updating array elements is unachievable. Such processes are part of all selected benchmarks, meaning the presence of this issue is aggravating or rendering impossible an implementation. Later is true for FFT. Moreover, there is an additional limitation directly impacting a selected benchmark. Currently, DaphneDSL does not support strings; thus, Sort is not implementable. Consequently, the list of rewritten benchmarks is reduced to HPL-AI, STREAM, PTRANS, and Mandelbrot.

4.5 Implementation

The DaphneDSL benchmarks use several metrics to evaluate execution performance. In addition, those are collected for the distinct elements performed by each application. Table 4.2 lists the program parts for each benchmark together with the used metric.

4.5.1 HPL-AI

The HPL-AI mixed precision benchmark [21] uses LU decomposition to solve a linear system $Ax = b$ with A and b being of size $n \times n$ and $n \times 1$, respectively. The unique feature is that solving is done in single precision. A and b in double precision are cast to 32-bit representations before executing the solving procedure. Subsequently, the single precision solution vector x_{32} is converted back to 64-bit. The resulting x_{64} is not the exact solution to the initial double precision problem. To reinstate accuracy, HPL-AI uses the Generalized Minimal Residual Method [7]. GMRES solves $Ax = b$ based on iterative adapting the initial

guess, here the x_{64}.

GMRES and LU-decomposition can not be adapted in DaphneDSL as the methods are based on iterative computation. Nonetheless, HPL-AI was implemented. The LU-decomposition was simply replaced by `solve()` from DaphneDSL, whereas GRMES was substituted with a different procedure. By default, the reference implementation uses a strictly diagonally dominant symmetric matrix A. In such a case, the system can be sequentially solved with Gauss-Seidel [8] based on an initial guess of x. Given a linear system $Ax = b$, A can be decomposed as $A = L + D + U$ with L the lower triangular part, D the diagonal, and U the upper triangular part. Then it holds that

$$(L + D)x_{k+1} = b - Ux_k \tag{4.1}$$

$$\Rightarrow x_{k+1} = (L + D)^{-1}(b - Ux_k), \tag{4.2}$$

with $(L + D)^{-1}$ being the inverse of $(L + D)$ and k the number of the iteration. It is noteworthy that Gauss-Seidel is still based on iterations but, in contrast to GMRES, does not update individual entries of x but rather x as a whole in each update cycle. This is feasible in DaphneDSL. As no inverse function is available in DaphneDSL, `solve()` is use to get x_{k+1} in equation 4.1.

The entire script is displayed in Code C.17. Apart from the difference regarding the last part, the implementation is in accordance with the reference benchmark. However, DaphneDSL uses matrix functionality, unlike in the original code, where the computations are conducted by means of loops over array elements. Similar to the reference implementation, the DaphneDSL benchmark measures performance via execution time. Two arguments are passed to the benchmark: the linear system's size n and the maximum number of iterations allowed for Gauss-Seidel max_iter.

4.5.2 STREAM

STREAM is executing four basic vector operations. Copy, scale, add, and so-called triad, a combination of scale and add. The uniqueness is the way those are performed. On each call, the vectors a, b and c are used in all four operations, meaning the next calculation is based on the results from the first. The operations are

$$\begin{aligned} c &= a && \text{copy,} \\ b &= c * scalar && \text{scale,} \\ c &= a + b && \text{add,} \\ a &= b + c * scalar && \text{triad,} \end{aligned}$$

with a=1, b=2, c=0 and $scalar$=0.42 as initial values [20]. The four operations are executed multiple times, each depending on the result of the previous iteration. Performance is measured with the average, minimum and maximum execution time calculated over all repetitions, not including the first. Furthermore, the throughput of each operation is measured based on the array size and the minimal execution time. The array size is determined by the number of elements in each vector multiplied by eight, the size of a 64-bit float.

The implementation can be found in Code C.18. The arguments are the desired number of iterations *ntimes* together with *array_elements*, the size of *a*,*b* and *c*.

In the same way as the reference implementation, the DaphneDSL version contains a verification. Given the fact that the result is deterministic on the number of iterations, the final values can be tested for correctness. Equal to HPL-AI, STREAM implementation is based on matrix operations other than the reference benchmark. This distinction might question comparability.

4.5.3 Mandelbrot

The Mandelbrot set [43] is defined as the points C in the complex plane, for which the recursively defined series

$$z_{k+1} = z_k^2 + C \tag{4.3}$$

with $z_0 = 0$ is bounded under a maximal number of iterations k_max. The visualization of the Mandelbrot set in Figure 4.1 is based on coloring the pixels in the complex plane, where the x-axis represents the real number and the y-axis the imaginary unit of the complex number C. The colors symbolize how quickly equation 4.3 diverges. An integer value called count [43] represents the largest k such that $|z_k| <= 2$. Different colors represent different counts, where the black area shows complex numbers remaining below the limit for a given k_max.

Speaking of the reference benchmark [27], Mandelbrot is very interesting in relation to scheduling as each pixel in the complex plane is checked for going beyond the limit. As done for each pixel, the calculation for a point is stopped on reaching the boundary. However, iterations on matrix elements are not possible in DaphneDSL. In addition, there

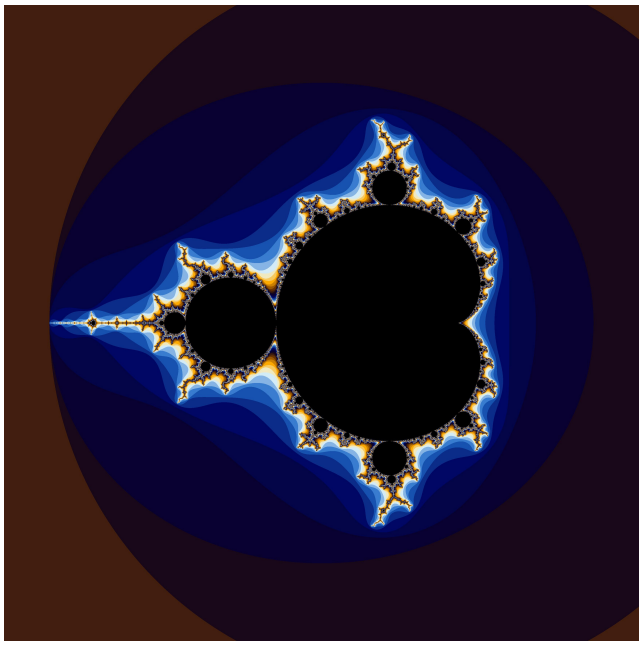

Figure 4.1: Visualization of the Mandelbrot set for 3500×3500 pixel complex plane with real numbers x=[-2.25, +1.25], imaginary unit y=[-1.75, +1.75] and maximum number of iterations k=2'000. Counts computed with DaphneDSL, visualization with Python.

is no current way to represent complex numbers. Consequently, the implementation is very different from the reference benchmark.

Firstly, the complex plane has to be specified with two matrices, one for the real part and the second for the imaginary unit. The real part matrix, for instance, has the same value on each row of one column representing the x-axis. Combined with the y-axis, those two matrices represent a pixel in the complex plane, and complex number operations are defined accordingly. Secondly, as there is no way to update elements of an array in a loop, all calculations are based on matrix operations and not on pixel-wise iterations. Lastly, for the same reason, all pixels must be evaluated in each iteration. In fact, that represents a violation of the underlying idea of the Mandelbrot benchmark, as a calculation for a pixel is not stopped when the complex number exceeds the limit.

Despite those shortcomings, Mandelbrot is implemented in DaphneDSL (Code C.19). The script takes two arguments. The number of Pixels n in each direction and a file f_set where the matrix with the calculated counts is written to. The axis limits (x=[-2.25, +1.25], y=[-1.75, +1.75]) and the maximum number of iterations (2'000) are set as to match the reference implementation. For performance measurement, the execution time is reported. Verification is done by means of visualizing the Mandelbrot set with Python based on the exported counts.

4.5.4 PTRANS

The writing of this benchmark is straightforward, given that elements of arrays are unchangeable within loops. This limitation leaves no alternative as to use the provided transpose function. In other words, the essential part of PTRANS in DaphneDSL narrows down to one line. This contrasts the reference benchmark with more than 1'600 lines of code

Table 4.2: DaphneDSL benchmarks: The program parts together with the reported metric.

Benchmark	Program part	Metric
HPL-AI	Initialization	Execution time (s)
	Convert to single	Execution time (s)
	Solve single	Execution time (s)
	Convert to double	Execution time (s)
	Gauss-Seidel	Execution time (s); Iterations needed
STREAM	Initialization	Execution time (s)
	Copy n times	Avg, min, max execution time (s); Throughput (GB/s)
	Scale n times	Avg, min, max execution time (s); Throughput (GB/s)
	Add n times	Avg, min, max execution time (s); Throughput (GB/s)
	Triad n times	Avg, min, max execution time (s); Throughput (GB/s)
Mandelbrot	Initialization	Execution time (s)
	Iteration	Execution time (s)
PTRANS	Initialization	Execution time (s)
	Iteration	Execution time (s)

(Table 4.1). The PTRANS routine in DaphneDSL is provided in Code C.20. The size of the square matrix to be transposed can be set with the argument n. In order to guarantee the correct result, the column sums of the input array are compared to the row sums of the resulting matrix. Execution time is measured and reported as a performance metric.

5
Experiments and Results

The performance evaluation of the DaphneDSL implementation against the reference benchmark is conducted by a design of factorial experiments. However, executing the reference implementation of PTRANS and HPL-AI on the computing system was impossible. Therefore, those two are not compared with their corresponding DaphneDSL benchmark.

5.1 Experiments

5.1.1 Scheduling Techniques

DAPHNE supports twelve different task partitioning schemes. A brief description is presented below. The full details, including formulas to determine the chuck size, are available in [14].

- Block Static (**STATIC**): Straightforward technique dividing tasks into chunks of equal size.
- Modified static (**MSTATIC**): Similar to STATIC, but using 4 times the number of chunks to divide tasks.
- Dynamic self-scheduling (**SS**): Chunk size is one task.
- Modified fixed-size chunk (**MFSC**): Modified version of fixed size self-scheduling (FSC). FCS assumes to know execution time before execution. MFSC does not require this profiling information [39].
- Guided self-scheduling (**GSS**): To balance execution among all units, GSS assigns decreasing chunk sizes.
- Trapezoid self-scheduling (**TSS**): Similar to GSS assigning decreasing chunk sizes but using a linear function to decrement chuck sizes.
- Factoring (**FAC2**): FAC assigns remaining loop iterations in batches of equally-sized chunks. FAC2 is a modification only assigning half of the remaining iterations.

- Trapezoid factoring self-scheduling(**TFSS**): Combines characteristics of FAC and TSS, schedules loop iterations in batches of equally-sized chunks like FAC, and decreases chunk size linearly like TSS.
- Fixed-increase self-scheduling (**FISS**): Using an increasing chuck size pattern.
- Variable-increase self-scheduling (**VISS**): A technique similar to FAC2 but increasing chunk sizes instead of decreasing them.
- Performance loop-based self-scheduling (**PLS**): The first part of a loop is scheduled statically, whereas the second part uses GSS. The amount of iterations for static scheduling is determined by PLS utilizing the static workload ratio, dividing the minimum by the maximum iteration execution time.
- Probabilistic self-scheduling (**PSS**): Scheduling of the number of iterations based on remaining iterations and the number of processors expected to be available in the future [39].

An illustrative list of chunk size patterns resulting from using different calculation techniques can be found in [14], Table 2. All those partitioning schemes are representatives of non-adaptive dynamic loop self-scheduling (DLS) techniques. They require certain information before application execution. In contrast to adaptive scheduling, they do not adjust scheduling decisions based on newly obtain information while executing [14]

5.1.2 Computing System

The performance evaluation is conducted on a small high-performance computing cluster at the University of Basel, the miniHPC [31]. The cluster serves two purposes. First, to offer students an instrument to achieve high-performance computations as part of teaching parallel programming, and second, an experimental platform to conduct scientific investigations in HPC. The miniHPC has 30 nodes of 4 different types. The experiments are executed on 22 computing nodes. A node has 2 Intel Xeon E5-2640 v4 CPUs. Each CPU offers 10 cores, 64 GB RAM, and 25 MB level 3 cache.

5.1.3 Design of Factorial Experiments

The design of factorial experiments is presented in Table 5.1. The applications tested are the four DaphneDSL implementations alongside two original benchmarks. The OpenMP versions of Mandelbrot [27] and STREAM [36] are adapted to be useable with different scheduling techniques, namely STATIC, SS and GSS. All 12 available run time partition schemes were included in the experiments for the DAPHNE implementations. The scalar coefficient for STREAM is modified to match the DaphneDSL implementation and to be equal to the one used in the current version of STREAM [20]. The reference applications are compiled using Intel compiler 2021.4.0.

The parameters play an important part, notably the problem size, which is the number of array elements used for each benchmark. One criterion was to use enough entries to exceed the available cache size. This is to assure main memory has to be used rather

Table 5.1: Design of factorial experiments resulting in a total of 7'380 experiments.

Factor	Value	Properties
Application	DaphneDSL STREAM	array_elements = 250′000 | 30M | 130M ntimes = 10 LOC: 263
	Reference STREAM	N = 250′000 | 30M | 130M NTIMES = 10 LOC: 586
	DaphneDSL Mandelbrot	n = 500 × 500 | 3′500 × 3′500 n_iter = 2′000 LOC: 141
	Reference Mandelbrot	Pixels = 500 × 500 | 3′500 × 3′500 max iterations/pixel = 2′000 LOC: 317
	DaphneDSL HPL-AI	n = 500 × 500 | 3′500 × 3′500 max_iter = n LOC: 221
	DaphneDSL PTRANS	n = 500 × 500 | 3′500 × 3′500 LOC: 104
Scheduling techniques	Non-adaptive dynamic loop self-scheduling	All: STATIC, SS, GSS Additional DAPHNE: MSTATIC, MFSC, TSS, FAC2, TFSS, FISS, VISS, PLS, PSS
Metrics	Program part performance	Execution time (s)
Computing system	1 miniHPC node	2 CPU Xeon E5-2640v4, 2.4GHz; 10 cores, 64GB RAM, 25 MB L3 cache each
Computing resources	Cores	4, 8, 20
Validity	Repetitions	20

than only the fast, accessible cache, in other words, to guarantee cache misses. Given the fact that one miniHPC node provides 50 MB of level three cache, the dimensions of the matrices in Mandelbrot, PTRANS and HPL-AI were set to $3'500 \times 3'500$, with eight bytes per entry, which is equal to an array size of roughly 93 MB. Alongside this "large" problem, a "small" problem size of 500×500 is part of the factorial properties. The idea is to analyze the behaviour when using arrays fitting into the cache. Regarding STREAM, the recommendation is to use data structured at least four times as large as the available cache ([36], code comments). Moreover, to meet the time calibration output of 20 clock ticks, the array size must be half of a chip's throughput capacity during 200 milliseconds. The suggestion is to use matrices with 1 GB, or 128M entries [36]. Therefore, vectors with sizes 30M and 130M are used for STREAM. The small problem size is set to 250'000.

The experimental runs are conducted using 4, 8 and 20 cores. In total, there are 369 factorial combinations. In order to increase the validity of the results, each experiment is repeated 20 times leading to an overall number of 7'380. For each experiment, the execution time in seconds of the corresponding program part (see Table 4.2) serves as a performance evaluation metric.

5.2 Results

Only two reference benchmarks were executable on miniHPC. Hence, solely Mandelbrot and STREAM can be evaluated against their reference implementation. Moreover, an overall assessment of the DaphneDSL implementations was part of the analysis.

5.2.1 DaphneDSL vs. Reference

5.2.1.1 Mandelbrot

The results for DaphneDSL and the reference benchmark are shown in Figure 5.1. Displayed is the total execution time, including all program parts (i.e., initialization and iteration) for both implementations and various experiment settings. From the different scales of the y-axis, it is evident that the newly written DaphneDSL applications are outperformed by the original. This result is no surprise, since the DaphneDSL implementation performs all iterations, whereas the reference benchmark stops once a complex pixel passes the stability bound.

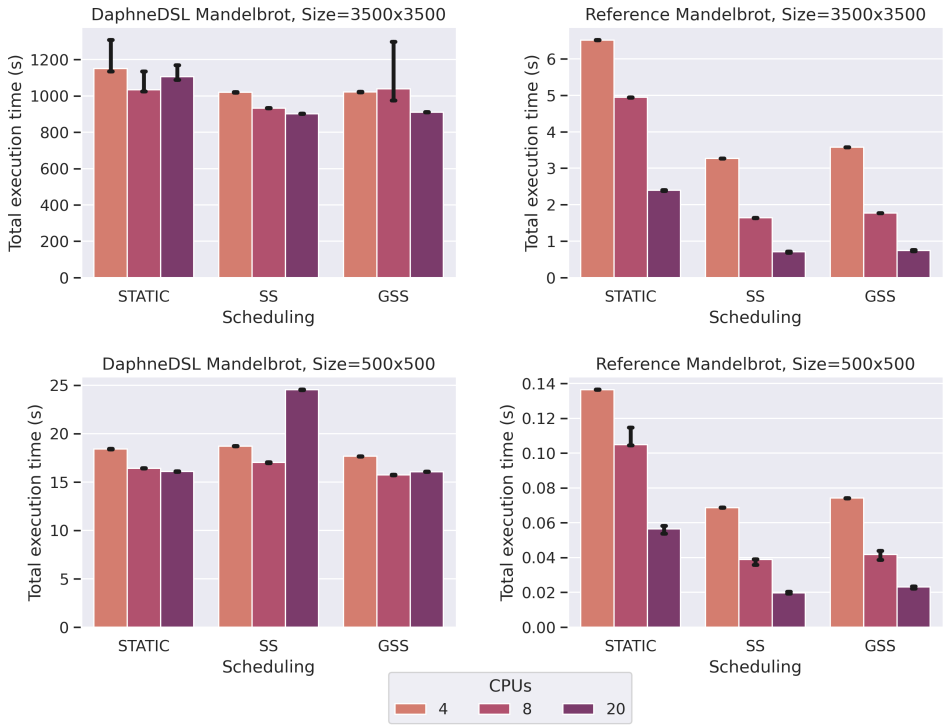

Figure 5.1: Mandelbrot benchmark: Total (i.e., including all program parts) execution time comparison between DaphneDSL implementation (left) and the original one (right) for different problem sizes. The maximum number of iterations is set to 2'000 in both cases. Colors according to the number of CPUs used. The bar height represents the median of 20 repetitions. The error is the 95% confidence intervals computed with bootstrapping. Plots created with Pythons Seaborn and Matplotlib libraries.

Controlling for this unavoidable dissimilarity is possible by comparing the number of actually performed iterations. The output of DaphneDSL Mandelbrot produces a matrix containing the counts, the number of iterations before a complex number exceeds the

limit. The counts plus 1 equals the number of iterations for each pixel performed by the reference implementation. Comparing the counts with the total number of computations of the DaphneDSL code, gives an understanding of the amount of unnecessary loop executions done by DAPHNE. The ratio of needed and maximum number of iterations for both problem sizes is roughly 13%. In other words, 87% of DAPHNE loop executions are dispensable.

However, even when naively correcting the execution time of DaphneDSL Mandelbrot for needless iterations, the values are still higher than the reference implementation. Admittedly, this is not a valid procedure as the execution time is driven by more than just a pure number of iterations (e.g., cache misses, scheduling). Nevertheless, it decreases the order of magnitude the reference benchmark outperforms the DaphneDSL implementation. In conclusion, our implementation of Mandelbrot appears less performing than the reference implementation. However, the comparability of those two is not given.

5.2.1.2 STREAM

The outcome of the experiments is presented by employing a heatmap and can be seen in Figure 5.2. The average execution time of the STREAM iterations with its four operations is displayed. The number in the squares is the median over the 20 experiments for different factor combinations. From the blue-colored tiles, it can be seen that DaphneDSL outperforms the reference implementation, at least for add, copy, and scale. Conversely, the reference is faster for triad, especially in combination with dynamic self scheduling (SS). Interestingly, DaphneDSL STREAM becomes slower with an increasing number of CPUs when using SS. This is observed for the more extensive array with 130M entries (Figure B.4). In fact, not all experiments finished in the allowed amount of time. In particular, the combination of DaphneDSL, dynamic self scheduling, and 20 CPUs resulted in an execution time exceeding the limit of three hours, leading to cancellation by the batch scheduling system on miniHPC. A reason is presented in Subsection 5.2.2.

To return to the subject. STREAM shows inconclusive results. DaphneDSL has, in most cases, a far better performance regarding execution time than the reference implementation. For copy, scale, and add, vectorized execution by DAPHNE is faster than parallel loop processing of OpenMP. The opposite is observed in the case of triad. Interestingly, the execution time for the reference benchmark is similar for all program parts. Therefore, the change in performance ranking is related to DAPHNE. The STREAM results for the small problem size in Figure B.2 show the same pattern, leading to the conclusion that this is not caused by an increased number of cache misses, as the small problem size fits into the cache. Moreover, this behaviour is independent of the used scheduling technique. Another explanation, the used function in DaphneDSL is not vectorized, meaning not executed parallel. A known example is `solve()`. Given that add and scale are fast, there is no reason to assume that triad is not executed in a vectorized manner. Thus, the cause remains unknown. The not conclusive results underline the statement that the comparability of the DaphneDSL implementation and the reference benchmark is questionable. Both have distinct ways to enable parallelism; hence a straightforward interpretation of the findings is inappropriate.

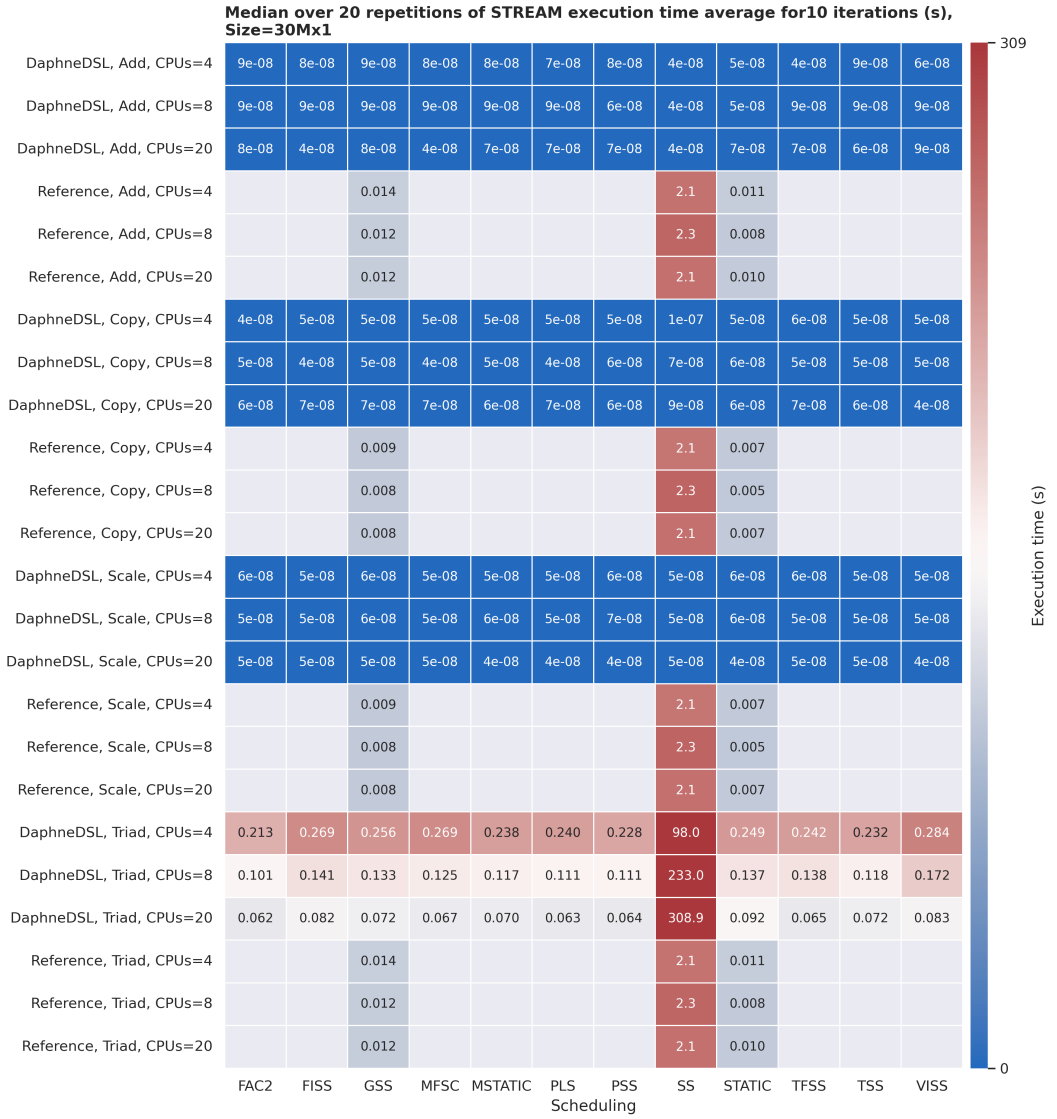

Figure 5.2: STREAM benchmark with array size 30Mx1: Number in squares is the median over 20 repetitions of the average execution time in seconds when executing STREAM with NTIMES = 10. Each row represents one STREAM operation (e.g., add, triad) for either the DaphneDSL or the reference implementation together with the CPUs used. Coloring according to execution time. Boxes without a number represent non-existing factor combinations. Plots created with Pythons Seaborn and Matplotlib libraries.

5.2.2 General DAPHNE Performance

To complement the result section, the performance of the four DaphneDSL benchmarks is analyzed further. The heatmaps with execution times for all factor combinations are to find in Figures B.3, B.4, B.5 and B.6. An illustrative example is depicted in Figure 5.3. Two points need to be emphasized. First, the execution time of DaphneDSL seems not driven by the number of CPUs used. The only reason for such behaviour is the small share of parallel execution time. In other words, the sequential time, as well as the overhead of scheduling, dominates the execution time resulting in no gain from using additional processing units; on the contrary, the performance has deteriorated. That is true for HPL-AI, where solving the

system at single precision and Gauss-Seidel use DaphneDSLs' `solve()`, a function known to be not vectorized. Second, self scheduling is performing worse the more CPUs are used. This observation is not new and has been encountered running STREAM. SS has the highest scheduling overhead of the available techniques [14]. With more CPUs, more scheduling is needed; consequently, the large overhead of self scheduling negatively affects performance.

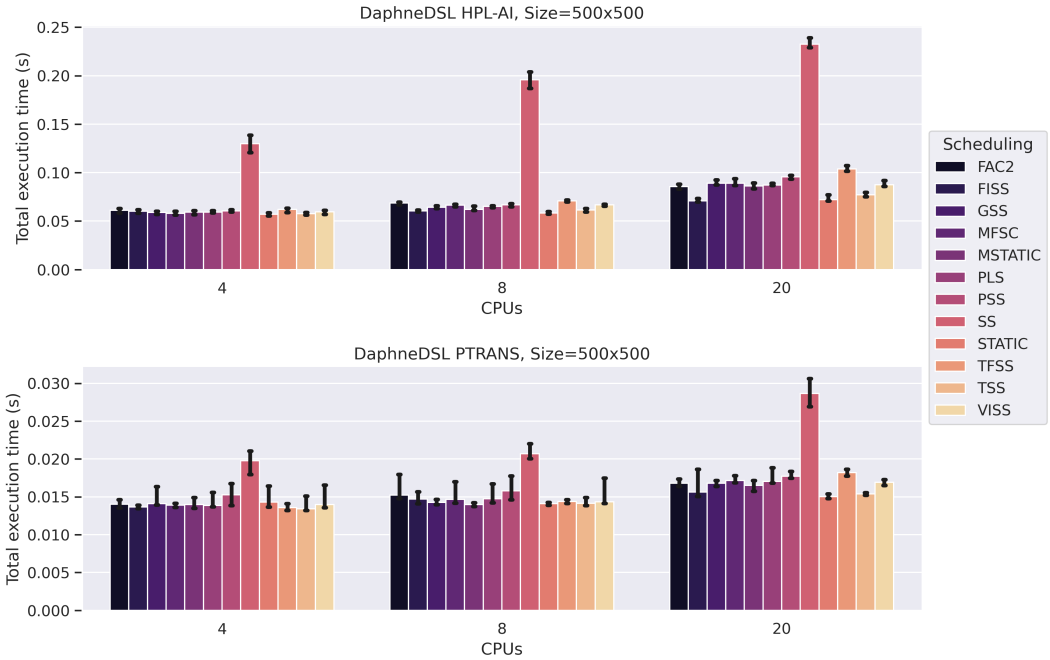

Figure 5.3: DaphneDSL small problem size: Total execution time including all program parts of HPL-AI (top) and PTRANS (bottom) for different CPUs. Colors according to the scheduling technique. The bar height represents the median of 20 repetitions. The error is the 95% confidence intervals computed with bootstrapping. Plots created with Pythons Seaborn and Matplotlib libraries.

6 Conclusion

The main contribution of this thesis is the implementation of benchmarks in DaphneDSL. Referring back to the question from the introduction, the revealed issues and limitations make it difficult or, in some instances, impossible to rewrite a benchmark application in DaphneDSL per a reference implementation. Furthermore, the inconclusive results for performance evaluation show that a direct comparison with the reference benchmark is at least questionable. The reason is the distinction in parallelism. DaphneDSL is intended to work with matrices using vectorized execution, whereas all considered benchmarks rely on parallel loop execution. Even in the absence of all issues, this circumstance prevents a comparable implementation of a reference benchmark with DaphneDSL.

6.1 Future Work

Further work is certainly required the moment some of the issues are resolved. Implementing Mandelbrot based on loops allows for verification that DaphneDSL is indeed slower than the reference benchmark, at least in a single CPU experiment without parallelism. So far, the results are deteriorated by the number of unnecessary iterations executed by the new implementation. At any rate, the Mandelbrot implementation has to be rewritten once parallel loop processing is supported in DAPHNE.

A more exciting direction for future work is the consideration of more complex reference benchmarks. This thesis has focused on applications testing basic tasks. The next logical step is to drop this criterion. As DAPHNE aims to provide means for IDA pipelines, candidates are the ML topics from HiBench or the component benchmarks from BigDataBench.

Bibliography

[1] ANother Tool for Language Recognition (ANTLR). https://www.antlr.org/. Accessed: 2022-07-02.

[2] Baidu Research DeepBench. https://github.com/baidu-research/DeepBench. Accessed: 2022-04-12.

[3] BenchCouncil: BigDataBench. https://www.benchcouncil.org/BigDataBench/index.html#Benchmarks. Accessed: 2022-05-02.

[4] BenchCouncil: BigDataBench 5.0 User Manual. http://www.benchcouncil.org/BigDataBench/files/BigDataBench5.0-User-Manual.pdf. Accessed: 2022-05-02.

[5] BenchCouncil: BigDataBench Component Benchmark Reference Implementation. https://github.com/BenchCouncil/BigDataBench_V5.0_BigData_ComponentBenchmark. Accessed: 2022-06-16.

[6] BenchCouncil: BigDataBench Micro Benchmark Reference Implementation. https://github.com/BenchCouncil/BigDataBench_V5.0_BigData_MicroBenchmark. Accessed: 2022-06-16.

[7] N. Black and S. Moore. Generalized Minimal Residual Method. In MathWorld - A Wolfram Web Resource, created by Eric W. Weisstein. https://mathworld.wolfram.com/GeneralizedMinimalResidualMethod.html, . Accessed: 2022-06-16.

[8] N. Black and S. Moore. Gauss-Seidel Method. From MathWorld–A Wolfram Web Resource created by Eric W. Weisstein. https://mathworld.wolfram.com/Gauss-SeidelMethod.html, . Accessed: 2022-07-05.

[9] BLAS (Basic Linear Algebra Subprograms). https://www.netlib.org/blas/. Accessed: 2022-07-02.

[10] P. Damme, M. Birkenbach, C. Bitsakos, M. Boehm, P. Bonnet, F. M. Ciorba, M. Dokter, P. Dowgiallo, A. Eleliemy, C. Faerber, G. Goumas, D. Habich, N.Hedam, M. Hofer, W. Huang, K. Innerebner, V. Karakostas, R. Kern, T.Kosar, D. Krems, A. Laber, W. Lehner, E. Mier, M. Paradies, B. Peischl, G. Poerwawinata, S. Psomadakis, T. Rabl, P. Ratuszniak, A. Starzacher, P. Silva, N. Skuppin, B. Steinwender, I. Tolovski, P. Tözün, W. Ulatowski, Y. Wang, I. Wrosz, A. Zamuda, C. Zhang, and X. Zhu. Daphne: An open and extensible system infrastructure for integrated data analysis pipelines. In *Proceedings of the 12th Annual Conference on Innovative Data Systems Research (CIDR '22), Chaminade, USA, January 9-12*, 2022.

[11] DaphneDSL: Built-ins and Grammar. https://github.com/daphne-eu/daphne/tree/main/src/parser/daphnedsl. Accessed: 2022-07-02.

[12] DaphneDSL documentation (work in progress). https://github.com/daphne-eu/daphne/blob/118-doc-daphnedsl/doc/DaphneDSL.md. Accessed: 2022-07-02.

[13] J. Dongarra, M. Herouxand, and P. Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications*, 30, 08 2015.

[14] A. Eleliemy and F. M. Ciorba. A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems. *Journal of Computational Science*, 51:101284, 2021. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2020.101284. URL https://www.sciencedirect.com/science/article/pii/S1877750320305792.

[15] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye, H. Tang, Z. Cao, S. Zhang, and J. Dai. BigDataBench: A Scalable and Unified Big Data and AI Benchmark Suite. 2018. URL https://arxiv.org/abs/1802.08254.

[16] A. Ghazal, T. Rabl, M. Hu, F. Raab amd M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, June 2013*, page 1197–1208, 2013. URL https://doi.org/10.1145/2463676.2463712.

[17] High Performance Conjugate Gradients (HPCG) Benchmark. https://www.hpcg-benchmark.org/index.html. Accessed: 2022-04-24.

[18] High Performance Conjugate Gradients (HPCG) Benchmark Reference Implementation. https://github.com/hpcg-benchmark/hpcg/. Accessed: 2022-06-16.

[19] HPC Challenge, Innovative Computing Laboratory - University of Tennessee Knoxville. https://icl.utk.edu/hpcc/. Accessed: 2022-06-16.

[20] HPC Challenge Reference Implementation. https://github.com/icl-utk-edu/hpcc. Accessed: 2022-06-16.

[21] HPL-AI Mixed-Precision Benchmark. https://www.hpl-ai.org/doc/index/. Accessed: 2022-04-17.

[22] HPL-AI Reference Implementation. https://bitbucket.org/icl/hpl-ai/src/main/. Accessed: 2022-06-16.

[23] N. Ihde, P. Marten, A. Eleliemy, G. Poerwawinata, P. Silva, I. Tolovski, F. M. Ciorba, and T. Rabl. A survey of big data, hpc and machine learning benchmarks. In *Proceedings of the 13th Transaction Processing Council Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2021) of the 47th International Conference on Very Large Data Bases, Copenhagen, Denmark, August*, 2021.

[24] Intel HiBench Reference Implementation. https://github.com/Intel-bigdata/HiBench. Accessed: 2022-04-02.

[25] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davisand J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[26] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. 2005.

[27] Mandelbrot reference implementation by John Burkardt. https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_openmp/. Accessed: 2022-06-30.

[28] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. A. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. M. Hazelwood, A. Hock, X. Huang, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St. John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia. MLPerf Training Benchmark. 2019. URL http://arxiv.org/abs/1910.01500.

[29] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/ref.html. Accessed: 2022-04-21.

[30] Merriam-Webster.com. Benchmark. https://www.merriam-webster.com/dictionary/benchmark. Accessed: 2022-05-02.

[31] miniHPC - High Performance Computing Group. https://hpc.dmi.unibas.ch/en/research/minihpc/. Accessed: 2022-07-07.

[32] MLCommons. https://mlcommons.org/en/training-normal-20/. Accessed: 2022-07-10.

[33] MLCommons Reference Implementation. https://github.com/mlcommons. Accessed: 2022-04-25.

[34] V. Janapa Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. St. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. Tejusve Raghunath Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. MLPerf Inference Benchmark. 2019. URL http://arxiv.org/abs/1911.02549.

[35] Standard Performance Evaluation Corporation. https://www.spec.org/benchmarks.html#current. Accessed: 2022-04-16.

[36] STREAM reference implementation by John D. McCalpin. https://www.cs.virginia.edu/stream/FTP/Code/stream.c. Accessed: 2022-07-02.

[37] The DAPHNE Project. https://daphne-eu.eu/project/. Accessed: 2022-07-02.

[38] The DAPHNE Project: Language Design Specification. http://daphne-eu.eu/wp-content/uploads/2022/06/DAPHNE_D3.1_LanguageDesign_v1.2.pdf. Accessed: 2022-07-07.

[39] The DAPHNE Project: Scheduler Design for Pipelines and Tasks. http://daphne-eu.eu/wp-content/uploads/2021/11/Deliverable-5.1-fin.pdf. Accessed: 2022-07-07.

[40] The DAPHNE System. https://github.com/daphne-eu/daphne. Accessed: 2022-07-02.

[41] Transaction Processing Performance Council Express Benchmark BB (TPCx-BB). https://www.tpc.org/tpcx-bb/default5.asp. Accessed: 2022-04-24.

[42] Unified European Applications Benchmark Suite. https://repository.prace-ri.eu/git/UEABS/ueabs. Accessed: 2022-04-16.

[43] E. W. Weisstein. Mandelbrot Set. From MathWorld - A Wolfram Web Resource created by Eric W. Weisstein. https://mathworld.wolfram.com/MandelbrotSet.html. Accessed: 2022-06-30.

A Tables

A.1 Detailed Benchmark List

Table A.1: Benchmarks and their main scripts

Suite	Benchmark	Domain	Metrics	Scripts	LOC
HPL-AI	Solving Linear System	HPC ML	run time GFLOPs	hpl-ai.c	134
				matgen.c	94
				convert.c	26
				sgetrf_nopiv.c	68
				blas.c	214
				gmres.c	208
BigData Bench	Sort	BD	run time	gen_random_text.cpp[2]	214
				mpi_sort.cpp	101
				ExternSort.h	541
BigData Bench	Grep	BD	run time	gen_random_text.cpp[2]	214
				mpi_grep.cpp	274
BigData Bench	WordCount	BD	run time	gen_random_text.cpp[2]	214
				mpi_wordcount.cpp	299
BigData Bench	MD5	BD	run time	gen_random_text.cpp[2]	214
				mpi_md5.cpp	307
				md5.h	93
BigData Bench	RandSample	BD	run time	gen_random_text.cpp[2]	214
				mpi_randsample.cpp	309

Continued on next page

Table A.1: Benchmarks and their main scripts (Continued)

Suite	Benchmark	Domain	Metrics	Scripts	LOC
BigData Bench	FFT	BD	run time	generate-matrix-float.c	37
				generate-matrix-int.c	37
				change-tripleMatrix-T.c	52
				mpiFFT.ccp	83
HPC Challenge	DGEMM[1]	HPC	run time GFLOPs	tstdgemm.c	157
HPC Challenge	PTRANS[1]	HPC	run time GB/s	pdmatgen.c	602
				pdtrans.c	925
				pdmatcomp.c	85
HPC Challenge	Random Access[1]	HPC	G updates per second	MPIRandomAccess.c	906
				time_bound.c	546
				buckets.c	131
				pool.c	83
				heap.c	195
HPC Challenge	FFT[1]	HPC	run time GFLOPs	mpifft.c	252
				wrapmpifftw.c	147
				fft235.c	860
HPC Challenge	STREAM[1]	HPC	run time GB/s	stream.c	714
HPC Challenge	HPL[1]	HPC	run time GFLOPs	HPL_pddriver.c	301
				HPL_pdinfo.c.c	1167
				HPL_pdtest.c	455
HPCG	Conjugate-gradient	HPC	run time FLOP count GFLOPs GB/s	main.cpp	380
				GenerateProblem_ref.cpp	219
				GenerateCoarseProblem.cpp	111
				ComputeSPMV_ref.cpp	72
				ComputeMG_ref.cpp	64
				ComputeSYMGS_ref.cpp	104
				OptimizeProblem.cpp	107
				ReportResults.cpp	413

[1] Called with https://github.com/icl-utk-edu/hpcc/blob/main/src/hpcc.c

[2] Not including the list of 7'762 words: lda_wiki1w.voca

Note: LOC includes main scripts and header files containing methods. Shell code for execution and files providing data structures are omitted.

Source: [6, 18, 20, 22]

A.2 Issue List

Table A.2: List of DaphneDSL issues.

GitHub	Issue	Description	MWE	Type	Status
#350	Rbind row check	rbind() checking number of rows instead of columns	C.1	Bug	Closed
#186	Parsing $n-1$	Parsing only works if there is a space after the $-$, i.e., $n-$ 1	C.2	Bug	Open
#351	Printing with variables	Printing with calculated variables behaving differently than with assigned variables	C.3	Bug	Open
#352	Integer matrix multiplication	Integer matrix multiplication not working	C.4	Bug	Pull request
#353	Right indexing	Right indexing with variables in loops is different from outside and is not working in functions	C.5	Bug	Open
#354	Left indexing	Left indexing with variables in loops is not working properly	C.6	Bug	Open
#355	Return from nrow()	Retrun value from nrow() not usable in operation	C.7	Bug	Open
#356	Type casting matrix	Type casting of matrices not working	C.8	Bug	Closed
#371	Shape change in loops	Combining matrices or changing matrix elements in loops is not possible	C.9	Bug	Open
#393	Built-in functions sum() and mean()	sum() works column and row wise while mean() is only working column wise	C.10	Bug	Closed
#395	Built-in function stddev()	stddev() works only column wise	C.11	Bug	Open
#396	Built-in function var()	var() is not supported	C.12	Bug	Open
#357	typeof() function	Function returning the type of an object	C.15	Nice to have	Open
#358	Matrix literals	Simple way to create a not random value matrix	C.13	Nice to have	Closed
#203	Left scalar multiplication	Multiplying a matrix with a scalar from left is not supported yet. Only right multiplication is possible.	C.16	Nice to have	Open

B

Figures

B.1 Results

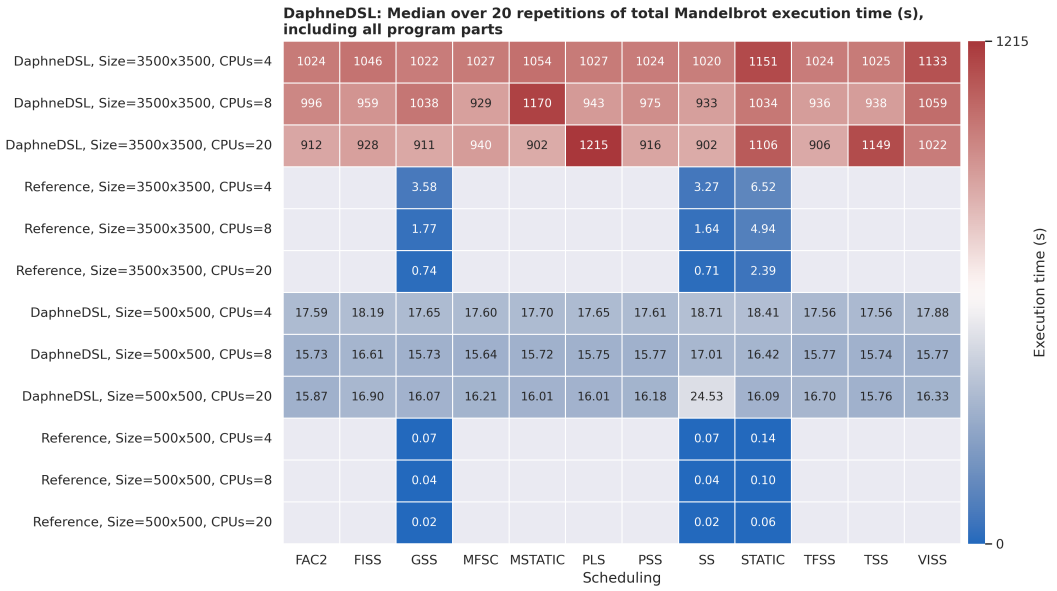

Figure B.1: Mandelbrot benchmark: Number in squares is the median over 20 repetitions of the total execution time including all program parts in seconds when executing Mandelbrot with 2'000 iterations. Each row represents a factor combination of benchmark, problem size and used CPUs. Coloring according to execution time. Boxes without a number represent non existing factor combinations. Plots created with Pythons Seaborn and Matplotlib libraries.

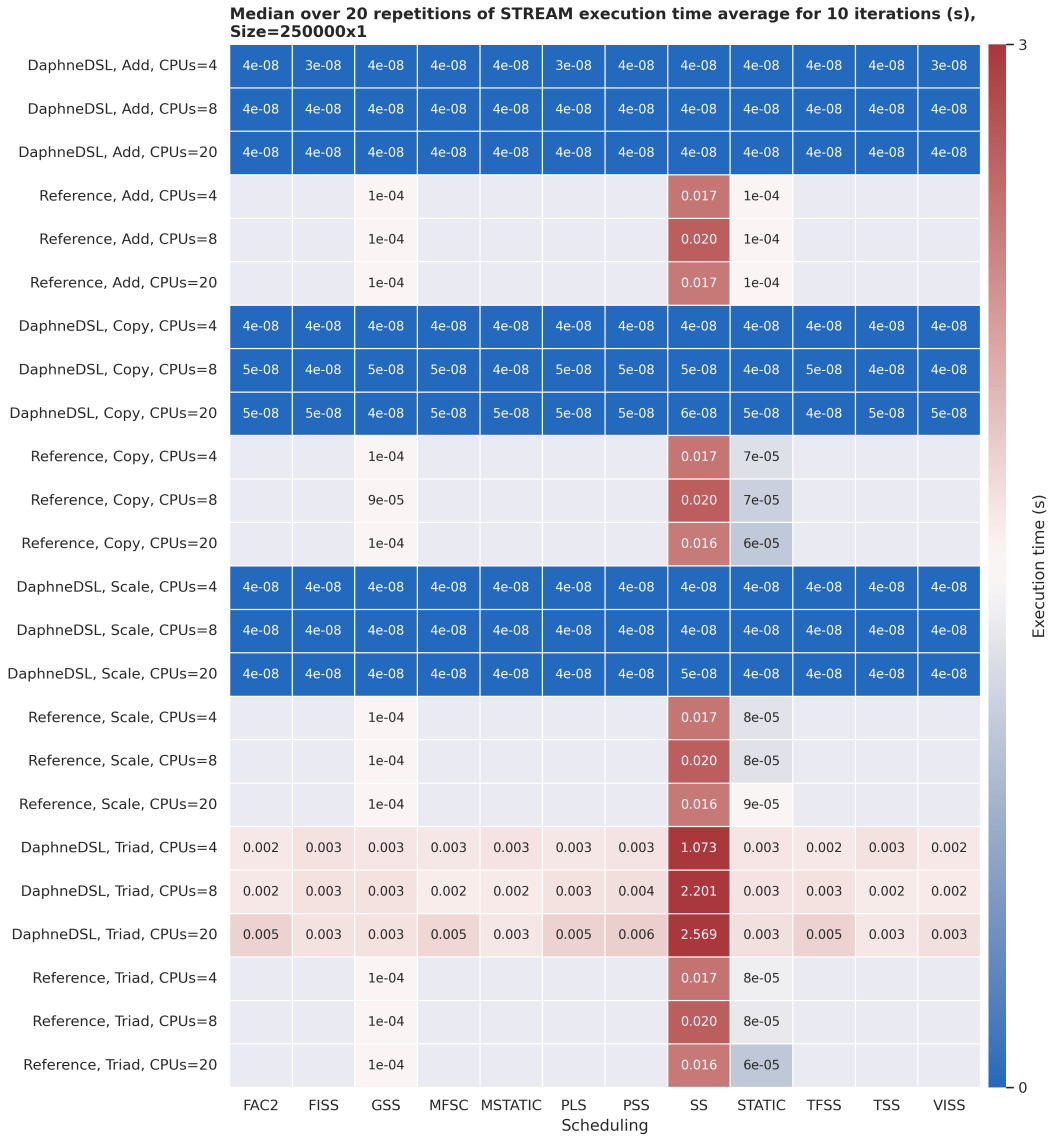

Figure B.2: STREAM benchmark with array size 250'000x1: Number in squares is the median over 20 repetitions of the average execution time in seconds when executing STREAM with NTIMES = 10. Each row represents one operation (e.g., add, triad) of STREAM for either the DaphneDSL or the reference implementation together with the CPUs used. Coloring according to execution time. Boxes without a number represent non existing factor combinations. Plots created with Pythons Seaborn and Matplotlib libraries.

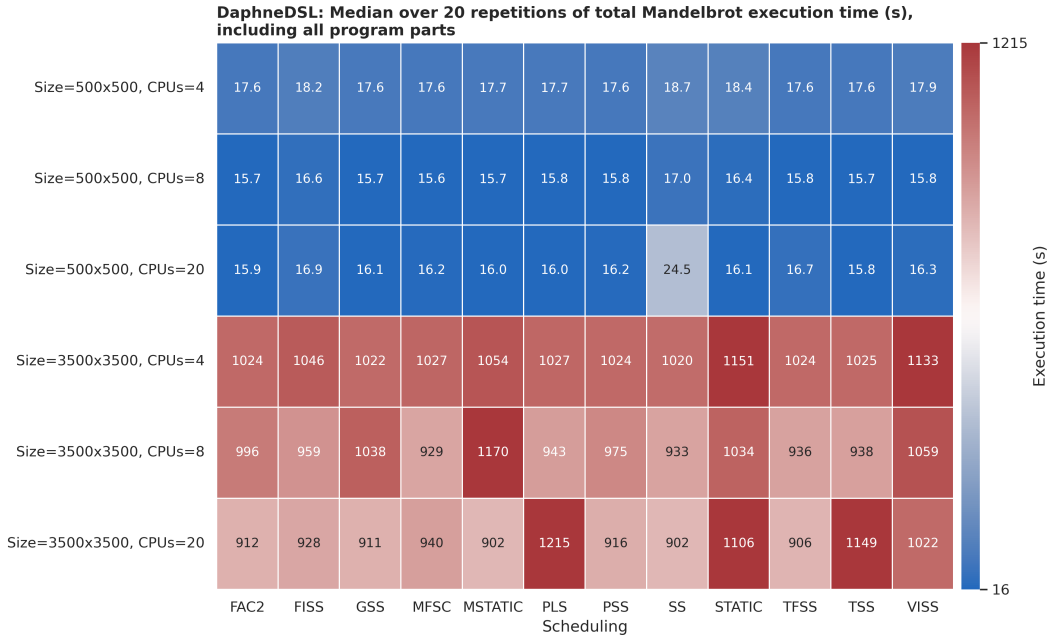

Figure B.3: Mandelbrot benchmark: Number in squares is the median over 20 repetitions of the total execution time in seconds including all program parts when executing with 2'000 iterations. Each row is a different factor combination of problem size and CPUs. Coloring according to execution time. Plots created with Pythons Seaborn and Matplotlib libraries.

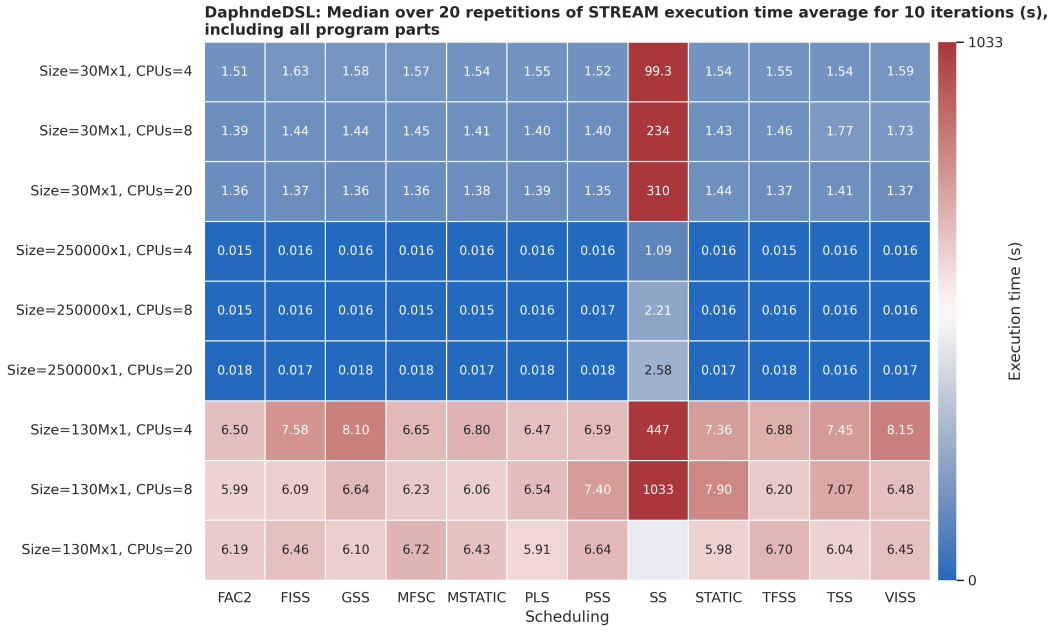

Figure B.4: STREAM benchmark: Number in squares is the median over 20 repetitions of the average execution time in seconds including all program parts when executing with NTIMES = 10. Each row is a different factor combination of problem size and CPUs. Coloring according to execution time. Empty boxes experiment did not finish. Plots created with Pythons Seaborn and Matplotlib libraries.

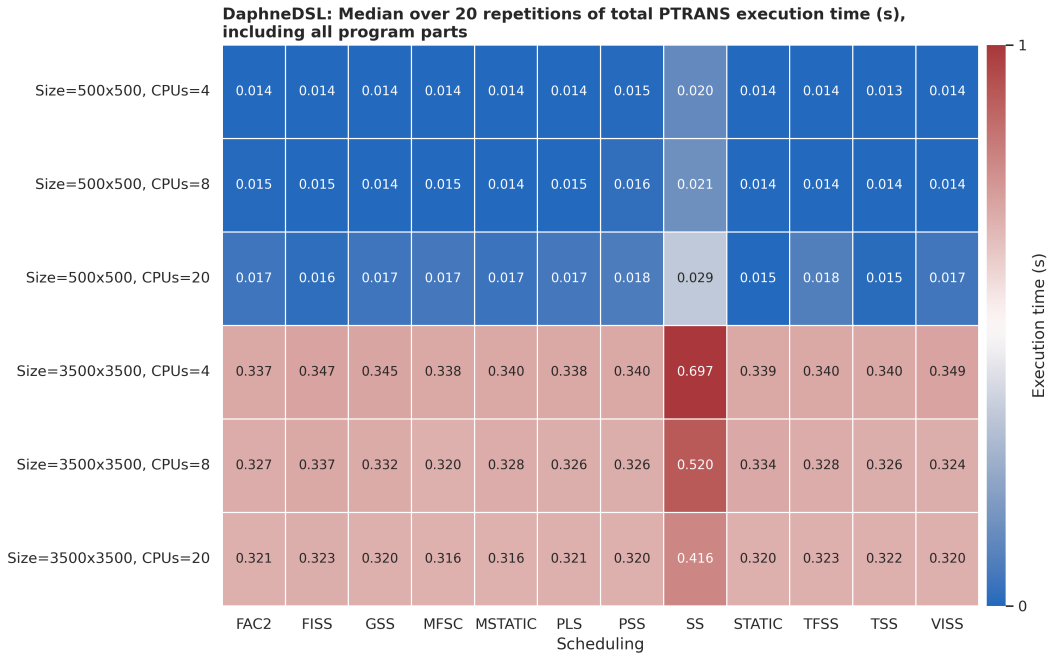

Figure B.5: PTRANS benchmark: Number in squares is the median over 20 repetitions of the total execution time in seconds including all program parts. Each row is a different factor combination of problem size and CPUs. Coloring according to execution time. Plots created with Pythons Seaborn and Matplotlib libraries.

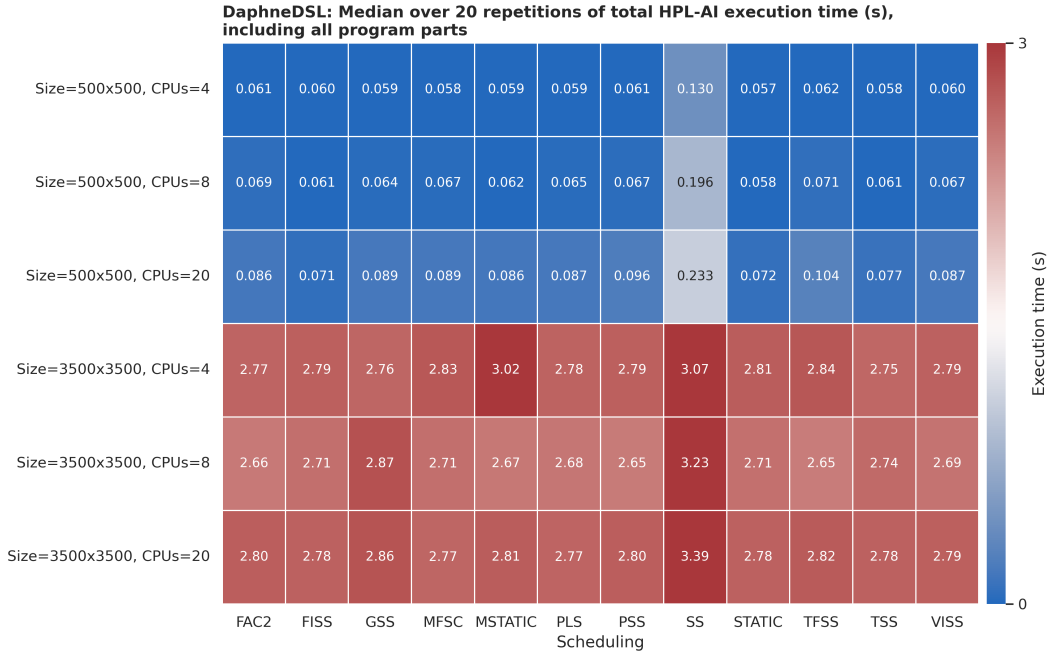

Figure B.6: HPL-AI benchmark: Number in squares is the median over 20 repetitions of the total execution time in seconds including all program parts. Each row is a different factor combination of problem size and CPUs. Coloring according to execution time. Plots created with Pythons Seaborn and Matplotlib libraries.

C
Code

C.1 DaphneDSL Issues

C.1.1 Bugs

Code C.1: Example to generate rbind() row check bug.

```
# Rbind is checking for number of rows instead of columns
r = rand(1,5,0,1,1,-1);
r2 = rand(1,5,1,2,1,-1);
r3 = rand(1,5,2,3,1,-1);
print(r);
print(r2);
print(r3);
r = rbind(r,r2);
print(r);
r = rbind(r,r3);
# Output:
# Pass error: shape inference:
# inferNumColsFromArgs() requires that arguments have the same number
# of columns, but there is one with 2 and one with 1 columns
```

Code C.2: Example to generate parsing $n-1$ bug.

```
# Parsing n-1
n=5;
print(n+1);
print(n - 1);
print(n- 1);
print(-1);
# not working
print(n-1);
```

Code C.3: Example to generate printing with variables bug.

```
# Printing with variables from computation
# working
time_convert = 200;
print("Time spend: " + time_convert + " seconds");
# this is not working correctly:
time_convert = 200 - 20;
print("Time spend:" + time_convert + " seconds");
# Output
# Time spend: 200 seconds
# 180Time spend: seconds
```

Code C.4: Example to generate integer matrix multiplication bug.

```
# Integer matrix multiplication
# working with double
d1 = reshape(seq(1.0,5.0,1.0),1,5);
print(d1);
d2 = reshape(seq(1.0,5.0,1.0),5,1);
print(d2);
print(d1 @ d2);
# not working with integer
i1 = reshape(seq(1,5,1),1,5);
print(i1);
i2 = reshape(seq(1,5,1),5,1);
print(i2);
print(i1 @ i2);
```

Code C.5: Example for right indexing bug.

```
# Right indexing with variables is working differently in loops/function than
m = rand(5,5,0.0,1.0,1,-1);
# working
i = 1;
print(m[i,i]);
# not working
for (i in 0:2) {
  print(m[i,i]);
}
# working, but not the most obvious way
for (i in 0:2) {
  pos_i = seq(i,i,1);
  print(pos_i);
  print(m[pos_i,pos_i]);
}
# not working in functions at all
def print_mat(mat :matrix) {
  print(mat[0,0]);
}
print_mat(m);
```

Code C.6: Example for left indexing bug.

```
# Left indexing with variables is working differently in loops/functions
# all working
A = rand(5,5,0.0,1.0,1,-1);
print(A);
A[0,0] = fill(10.0,1,1);
print(A);
B = rand(1,5,1.0,2.0,1,-1);
print(B);
A[4,:] = B;
print(A);
i = 3;
C = rand(1,4,2.0,3.0,1,-1);
A[i,1:] = C;
print(A);
# in loops
for (i in 0:2) {
  # working
  A = rand(5,5,0.0,1.0,1,-1);
  A[0,0] = fill(10.0,1,1);
  # working
  B = rand(1,5,1.0,2.0,1,-1);
  A[4,:] = B;
  # not working
  C = rand(1,4,2.0,3.0,1,-1);
  A[i,1:] = C;
  # Output: JIT session error: Symbols not found:
  # not working
  A[fill(i,1,1),1:] = C;
  # Output: Parser error: left indexing with positions
  # as a data object is not supported (yet)
}
# Workarround, we use ':' operator (inclusive:exclusive)
# Hence i:(i + 1) is just index i.
# working in functions
def change_element(A : matrix<f64>, i : si64) {
  C = rand(1,5,2.0,3.0,1,-1);
  A[i:(i+1),:] = C;
  return A;
}
A = rand(5,5,0.0,1.0,1,-1);
A = change_element(A,3);
print(A);
# In loops resulting in type/shape change issue #371
A = rand(5,5,0.0,1.0,1,-1);
for (i in 1:10) {
  A[i:(i + 1),0:1] = fill(0.0,1,1);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
}
```

Code C.7: Example to generate return from nrow() bug.

```
# result from nrow() is not usable in operations
m = rand(5,5,0.0,1.0,1,-1);
n = nrow(m);
print(n); # working
print(n+1); # not working
# Output:
# error: 'daphne.ewAdd' op operand #0 must be matrix of numeric
# or placeholder for an unknown type values or numeric, but got 'index'
# Workarround: Casting
n = as.si64(nrow(A));
```

Code C.8: Example to generate matrix type casting bug.

```
# Type casting matrices not working
A = rand(5,5,0.0,1.0,1,-1);
print(A);
B = as.matrix.f32(A);
# Output:
# JIT session error: Symbols not found:
#[ _cast__DenseMatrix_float__DenseMatrix_double ]
```

Code C.9: Example to generate shape change in loop error.

```
# shape change within loops
m = fill(0.0,1,1);
iter = 1;
while (iter < 11) {
  e = fill(0.0,1,1);
  m = rbind(m,e);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
  iter = iter + 1;
}
m = fill(0.0,1,1);
for (iter in 1:10) {
  e = fill(0.0,1,1);
  m = cbind(m,e);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
}
A = rand(5,5,0.0,1.0,1,-1);
for (i in 1:10) {
  A[i:(i + 1),0:1] = fill(0.0,1,1);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
}
```

```
# idea, use functions and then call with loops
def via_operator(row : si64, col : si64, A : matrix<f64>, value : f64) ->
    matrix<f64> {
  # Working with : operator
  A[row:(row + 1),col:(col + 1)] = fill(value,1,1);
  return A;
}
# This function was designed to not use left indexing at all but
# slicing into 4 blocks arround the value to replace
# and recombining it.
def via_bind(row : si64, col : si64, A : matrix<f64>, value : f64) -> matrix<
    f64> {
  n_row = nrow(A);
  n_col = ncol(A);
  # all left of it
  A_l = A[:,0:col]; # its exclusive, 0:2 cols 0:1
  # all right of it
  A_r = A[:,(col + 1):n_col];
  # all above
  A_a = A[0:row,col:(col + 1)];
  # all below
  A_b = A[(row + 1):n_row,col:(col + 1)];
  # value in 1x1
  value = fill(value,1,1);
  # putback together
  val_col = rbind(A_a,value);
  val_col = rbind(val_col, A_b);
  B = cbind(A_l,val_col);
  B = cbind(B, A_r);
  return B;
}
# Function calls working
A = rand(5,5,0.0,1.0,1,-1);
B = via_bind(1,2,A,2.1);
C = via_operator(1,2,A,5.1);
print(A);
print(B);
print(C);
# but not possible in loops
for (i in 1:10) {
  A = via_bind(1,2,A,2.1);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
}
for (i in 1:10) {
  A = via_operator(1,2,A,2.1);
  # Output: Pass error:
  # the type/shape of a variable must not be changed within the body of a for
    -loop
}
```

Code C.10: Example of sum() and only colum wise working mean().

```
# Built-in function issues
m = fill(2.0,5,2);
print(m);
# the built in functions sum and mean us arguments
# 0 = row wise, 1 = column wise, none = all
# sum working
print(sum(m)); # overall
print(sum(m, 0)); # row wise
print(sum(m, 1)); # column wise
# mean working
print(mean(m, 1)); # cloumn wise
# mean not working
print(mean(m)); # overall
print(mean(m, 0)); # row wise
```

Code C.11: Example for only working column wise stddev().

```
# Built-in function issues
m = fill(2.0,5,2);
print(m);
# the built in functions stddev us arguments
# 0 = row wise, 1 = column wise, none = all
# stddev working
print(stddev(m, 1)); # cloumn wise
# stddev not working
print(stddev(m, 0)); # row wise
print(stddev(m)); # overall
```

Code C.12: Example of not supported var() built-in function.

```
# Built-in function issues
m = fill(2.0,5,2);
print(m);
# the built in functions var is not working
# 0 = row wise, 1 = column wise, none = all
# var not working
print(var(m)); # overall
print(var(m, 1)); # cloumn wise
print(var(m, 0)); # row wise
```

C.1.2 Nice To Have

Code C.13: Example how to generate a matrix literal.

```
# create a simple matrix (like Matlab) with some desired values
# or at least vector, as reshape exists (like in R)
# The goal is to create a magic 5x5 matrix
# matlab
m = [17 24 1 8 15, 23 57 14 16, ...];
# R
v = c(17 2 4 1 8 15 23 5 7 14 16 ...);
m = reshape(m,5,2)
# or with matrix command
m = matrix(17 2 4 1 8 15 23 5 7 14 16 ...);
# Workaround:
# Generate 5x5 matrix...
mat = fill(0.0,5,5);
# ... and set each element
mat[0,0] = fill(17.0,1,1);
mat[0,1] = fill(24.0,1,1);
mat[0,2] = fill(1.0,1,1);
mat[0,3] = fill(8.0,1,1);
mat[0,4] = fill(15.0,1,1);
mat[1,0] = fill(23.0,1,1);
mat[1,1] = fill(5.0,1,1);
mat[1,2] = fill(7.0,1,1);
mat[1,3] = fill(14.0,1,1);
mat[1,4] = fill(16.0,1,1);
mat[2,0] = fill(4.0,1,1);
mat[2,1] = fill(6.0,1,1);
mat[2,2] = fill(13.0,1,1);
mat[2,3] = fill(20.0,1,1);
mat[2,4] = fill(22.0,1,1);
mat[3,0] = fill(10.0,1,1);
mat[3,1] = fill(12.0,1,1);
mat[3,2] = fill(19.0,1,1);
mat[3,3] = fill(21.0,1,1);
mat[3,4] = fill(3.0,1,1);
mat[4,0] = fill(11.0,1,1);
mat[4,1] = fill(18.0,1,1);
mat[4,2] = fill(25.0,1,1);
mat[4,3] = fill(2.0,1,1);
mat[4,4] = fill(9.0,1,1);
```

Code C.14: Implemented way to create matrix literals.

```
# Working version for matrix literals.
mat = [17, 24, 1, 8, 15, 6];
mat = reshape(mat,2,3);
print(mat);
# Output:
# DenseMatrix(2x3, int64_t)
# 17 24 1
# 8  15 6
```

Code C.15: Example how to use a typeof() function.

```
# type function for objects returning the type/typeof similar as in R
f = 5.0;
typeof(f); # --> f64
d = reshape(seq(1.0,5.0,1.0),5,1);
typeof(d1); # --> matrix double
i = reshape(seq(1,5,1),1,5);
typeof(i); # --> matrix int64
```

Code C.16: Example of left and right scalar multiplication.

```
# Left scalar multiplication
X = fill(1.0,10,10);
y = fill(2.0,10,1);
# not working
r = 1.5 * X @ y + 0.001;
# Output:
# JIT session error: Symbols not found:
# [ _ewMul__double__DenseMatrix_double__double ]
# JIT-Engine invocation failed:
# Failed to materialize symbols:
# { (main, { _mlir__mlir_ciface_main, _mlir_ciface_main, main, _mlir_main })
    }
# Program aborted due to an unhandled Error:
# Right scalar multiplication is working
r = X @ y * 1.5 + 0.001;
print(r);
```

C.2 Implementation

C.2.1 HPL-AI

Code C.17: HPL-AI benchmark implementation.

```
# ------------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# Modifications Copyright 2022 The DAPHNE Consortium
#
# ------------------------------------------------

# Script to perform HPL-AI Mixed-Precision benchmark for DAPHNE
# Details on the benchmark https://www.hpl-ai.org/doc/index/.
# Reference Impelmentation: https://bitbucket.org/icl/hpl-ai/src/main/

# Instead of GMRES which uses iterative updating of matrix elements
# (see https://www.netlib.org/templates/matlab/gmres.m), a functionality
# not supported yet (https://github.com/daphne-eu/daphne/issues/354),
# Gauss-Seidel is used which is working with matrix operations.
# (https://mathworld.wolfram.com/Gauss-SeidelMethod.html)

# GS is converging for strictly diagonally dominant square matrices
# which is given in the reference benchmark implementation.

# INPUT PARAMETERS:
# ------------------------------------------------
# NAME      TYPE        DEFAULT   MEANING
# ------------------------------------------------
# n         Integer     ---       Size of Linear System
# max_iter  Integer     ---       Maximum GS interations
# ------------------------------------------------
#
# OUTPUT:
# ------------------------------------------------
# NAME  TYPE    DEFAULT   MEANING
# ------------------------------------------------
```

```
# Reports execution time of each program part in seconds
# -----------------------------------------------

# -----------------------------------------------
# Global arguments
# -----------------------------------------------
# desired size
n = $n; # from command line input n=?
# maximum allowed iterations for GS
max_iter = $max_iter;

# Gauss-Seidel converges in n steps, thus set max_iter accordingly
if (max_iter > n) {
  max_iter = n - 1;
}

# calculate epsilon
epsilon = 1.0;
while ((1.0 + 0.5 * epsilon) != 1.0 ){
  epsilon = 0.5 * epsilon;
}
eps = epsilon / 2.0;
# threshold for iterative improvement as in original
threshold = 16.0;

# scaling factors for ns to second
ns_s = 1000000000.0;


# -----------------------------------------------
# Functions
# -----------------------------------------------

# Infinity norm: max(Row sum of absolut values of vector)
# outmost sum needed to make it f64
# else matrixf64 is returned
def norm(a : matrix<f64>) -> f64 {
  return sum(aggMax(sum(abs(a),0),1));
}


# Infinity norm: max(Row sum of absolut values of matrix)
def normMat(B : matrix<f64>) -> f64 {
  return sum(aggMax(sum(abs(B),0),1));
}

# -----------------------------------------------
# Routine
# -----------------------------------------------

print("==================");
print("Running HPL-AI");

time_start = now();
```

```
103
104 # Intialize Linear System
105 # strictly diagonally dominant square A
106 A_f64 = rand(n,n,as.f64(-0.5),as.f64(0.5),1,1);
107 row_sum = sum(abs(A_f64),0);
108 A_row_sum = diagMatrix(row_sum);
109 A_diag = A_f64 * diagMatrix(fill(1.0,n,n));
110 # the diagonal of the hpl-ai matrix is the sum of
111 # the absolute values of the off-diagonals on the same row.
112 A_f64 = A_f64 - A_diag + A_row_sum;
113 # vector b
114 b_f64 = rand(n,1,as.f64(-0.5),as.f64(0.5),1,2);
115
116
117 time_initialize = now() - time_start;
118 print("--------------------");
119 print("Initialization time:");
120 print(as.f64(time_initialize) / ns_s);
121 print("--------------------");
122
123 # Convert A and b to single
124 time_convert = now();
125
126 A_f32 = as.matrix.f32(A_f64);
127 b_f32 = as.matrix.f32(b_f64);
128
129 time_convert = now() - time_convert;
130 print("Convert to f32 time:");
131 print(as.f64(time_convert) / ns_s);
132 print("--------------------");
133
134
135 # Solving System
136 time_solve = now();
137
138 x_f32 = solve(A_f32,b_f32);
139
140 time_solve = now() - time_solve;
141 print("Solve f32 system time:");
142 print(as.f64(time_solve) / ns_s);
143 print("--------------------");
144
145
146 # Convert x to double
147 time_convert = now();
148
149 x_f64 = as.matrix.f64(x_f32);
150
151 time_convert = now() - time_convert;
152 print("Convert to f64 time:");
153 print(as.f64(time_convert) / ns_s);
154 print("--------------------");
155
156
```

```
157 # Restate precision:
158 # Gauss-Seidel
159 # Given: Ax = b
160 # with A = L + D + U it holds that
161 # x(k+1) = (D+L)^(-1) @ (b - U @ x(k))
162 # as transpose is not provided in DaphneDSL, we use solve
163 # (D+L) @ x(k+1) = b - U @ x(k)
164 time_solve = now();
165
166 # Make A = L + D + U
167 D = A_f64 * diagMatrix(fill(1.0,n,1));
168 L = A_f64 * lowerTri(A_f64, false, false);
169 U = A_f64 * upperTri(A_f64, false, false);
170 i = 0;
171
172 # checking convergence with the linear system error
173 # analogous to reference benchmark
174 resid = b_f64 - (A_f64 @ x_f64);
175 divisor = normMat(A_f64) * norm(x_f64) + norm(b_f64);
176 error = norm(resid) / divisor / as.f64(n) / eps;
177
178 # iterative improvement: Gauss-Seidel
179 while ((i < max_iter) && as.si64(error >= threshold)) {
180
181   DL = D + L;
182   Uxb = b_f64 - U @ x_f64;
183   x_f64 = solve(DL, Uxb);
184
185   i = i + 1;
186
187   resid = b_f64 - (A_f64 @ x_f64);
188   divisor = normMat(A_f64) * norm(x_f64) + norm(b_f64);
189   error = norm(resid) / divisor / as.f64(n) / eps;
190
191 }
192
193 time_solve = now() - time_solve;
194
195
196 if (i == max_iter) {
197   print("Gauss-Seidel did not coverge with:");
198   print(i);
199   print("iterations");
200   print("--------------------");
201   print("HPL-AI not finished.");
202 } else {
203   print("Gauss-Seidel converged with:");
204   print(i);
205   print("iterations in (s)");
206   print(as.f64(time_solve) / ns_s);
207   print("--------------------");
208   print("HPL-AI finished.");
209   print("--------------------");
210   print("Norm of residual r = b - Ax:");
```

```
  print("Double precision:");
  resid = b_f64 - (A_f64 @ x_f64);
  t = norm(resid);
  print(t);
  print("Single precision:");
  resid = b_f64 - (A_f64 @ as.matrix.f64(x_f32));
  t = norm(resid);
  print(t);
}

print("==================");
```

C.2.2 STREAM

Code C.18: STREAM benchmark implementation.

```
# ------------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# Modifications Copyright 2022 The DAPHNE Consortium
#
# ------------------------------------------------


# Script to perform STREAM benchmark for DAPHNE
# Details on the benchmark: https://icl.utk.edu/hpcc/
# Reference Impelmentation:
# https://github.com/icl-utk-edu/hpcc/blob/main/STREAM/stream.c
# Stand alone openMP version including a main method:
# https://www.cs.virginia.edu/stream/FTP/Code/stream.c


# INPUT PARAMETERS:
# ------------------------------------------------
# NAME          TYPE    DEFAULT MEANING
# ------------------------------------------------
```

```
# array_elements    Integer ---     Size of input vectors
# ntimes      Integer ---   Number of times to repeate the experiment
# -----------------------------------------------
#
# OUTPUT:
# -----------------------------------------------
# NAME  TYPE    DEFAULT   MEANING
# -----------------------------------------------
# Reports average, minimal and maximal execution time in seconds
# Rports throughput in GB/s
# -----------------------------------------------

# -----------------------------------------------
# Functions
# -----------------------------------------------

# -----------------------------------------------
# Global arguments
# -----------------------------------------------
# desired size
array_elements = $array_elements; # from command line input array_elements=?
ntimes = $ntimes; # from command line input ntimes=?

# scaling factors for ns to second and bytes to GiBs
GiBs = 1024.0 * 1024.0 * 1024.0;
ns_s = 1000000000.0;

# initial values from original implementation
ai = 1.0;
bi = 2.0;
ci = 0.0;
scalar = 0.42;

# set tolerance ('epsilon') as in multiLogReg.daphne
# example for GitHub DAPHNE repo
tol = 0.000001;

# -----------------------------------------------
# Routine
# -----------------------------------------------

print("====================");
print("Running STREAM");

time_start = now();

# generating vectors
# with values as in original implementation
a = fill(ai,array_elements,1);
b = fill(bi,array_elements,1);
c = fill(ci,array_elements,1);


# generate element result for later check
```

```
# first single ai
for (k in 1:ntimes) {
  # Copy
  ci = ai;
  # Scale
  bi = ci * scalar;
  # Add
  ci = ai + bi;
  # Triad
  ai = bi + ci * scalar;
}

# sum all ai
ai_sum = 0.0;
for (k in 1:array_elements) {
  ai_sum = ai_sum + ai;
}


# as indexing is not possible in DaphneDSL loops yet
# https://github.com/daphne-eu/daphne/issues/354
# we need variables insted of an array to measure time
time_copy_total = 0;
# as.si64(inf) returns negativ number, devide it with -2
# ui64(inf) would work, but min(a,b) needs same type
# and now() returns si64 which is the integer default type
time_copy_min = as.si64(inf) / -2;
time_copy_max = 0;
time_scale_total = 0;
time_scale_min = as.si64(inf) / -2;
time_scale_max = 0;
time_add_total = 0;
time_add_min = as.si64(inf) / -2;
time_add_max = 0;
time_triad_total = 0;
time_triad_min = as.si64(inf) / -2;
time_triad_max = 0;

time_initialize = now() - time_start;
print("--------------------");
print("Initialization time:");
print(as.f64(time_initialize) / ns_s);
print("--------------------");

for (k in 1:ntimes) {

  # as in original implementation, first iteration
  # is not measured, reset all time counters if k == 2
  if (k == 2) {
    time_copy_total = 0;
    time_copy_min = as.si64(inf) / -2;
    time_copy_max = 0;
    time_scale_total = 0;
    time_scale_min = as.si64(inf) / -2;
```

```
145     time_scale_max = 0;
146     time_add_total = 0;
147     time_add_min = as.si64(inf) / -2;
148     time_add_max = 0;
149     time_triad_total = 0;
150     time_triad_min = as.si64(inf) / -2;
151     time_triad_max = 0;
152   }
153
154
155   # Copy
156   time_start = now();
157   c = a;
158
159   # Copy timing
160   time_copy = now() - time_start;
161   time_copy_total = time_copy_total + time_copy;
162   time_copy_min = min(time_copy, time_copy_min);
163   time_copy_max = max(time_copy, time_copy_max);
164
165
166   # Scale
167   time_start = now();
168   b = c * scalar;
169
170   # Scale Timing
171   time_scale = now() - time_start;
172   time_scale_total = time_scale_total + time_scale;
173   time_scale_min = min(time_scale, time_scale_min);
174   time_scale_max = max(time_scale, time_scale_max);
175
176
177   # Add
178   time_start = now();
179   c = a + b;
180
181   # Add Timing
182   time_add = now() - time_start;
183   time_add_total = time_add_total + time_add;
184   time_add_min = min(time_add, time_add_min);
185   time_add_max = max(time_add, time_add_max);
186
187
188   # Triad
189   time_start = now();
190   a = b + c * scalar;
191
192   # Triad Timing
193   time_triad = now() - time_start;
194   time_triad_total = time_triad_total + time_triad;
195   time_triad_min = min(time_triad, time_triad_min);
196   time_triad_max = max(time_triad, time_triad_max);
197
198 }
```

```
199
200 # quality check
201 a_sum = sum(a);
202
203 # calculate throughput
204 # as in original use min time
205 # for copy and scale 2 arrays
206 copy_throughput = (as.f64(array_elements * 2 * 8) / GiBs) / as.f64(
        time_copy_min);
207 scale_throughput = (as.f64(array_elements * 2 * 8) / GiBs) / as.f64(
        time_scale_min);
208 # for add and triad 3 arrays
209 add_throughput = (as.f64(array_elements * 3 * 8) / GiBs) / as.f64(
        time_add_min);
210 triad_throughput = (as.f64(array_elements * 3 * 8) / GiBs) / as.f64(
        time_triad_min);
211
212 # print results if passed
213 if (abs(a_sum - ai_sum) <= tol) {
214
215   print("STREAM run results are correct.");
216   print("---------------------");
217
218   print("Copy execution time seconds: (avg, min, max):");
219   print((as.f64(time_copy_total) / (as.f64(ntimes - 1))) / ns_s); # skip
        first iteration
220   print(as.f64(time_copy_min) / ns_s);
221   print(as.f64(time_copy_max) / ns_s);
222   print("---------------------");
223
224   print("Copy throughput (GB/s):");
225   print(copy_throughput * ns_s); # convert to seconds
226   print("---------------------");
227
228   print("Scale execution time seconds: (avg, min, max):");
229   print((as.f64(time_scale_total) / (as.f64(ntimes - 1))) / ns_s);
230   print(as.f64(time_scale_min) / ns_s);
231   print(as.f64(time_scale_max) / ns_s);
232   print("---------------------");
233
234   print("Scale throughput (GB/s):");
235   print(scale_throughput * ns_s); # convert to seconds
236   print("---------------------");
237
238   print("Add execution time seconds: (avg, min, max):");
239   print((as.f64(time_add_total) / (as.f64(ntimes - 1))) / ns_s);
240   print(as.f64(time_add_min) / ns_s);
241   print(as.f64(time_add_max) / ns_s);
242   print("---------------------");
243
244   print("Add throughput (GB/s):");
245   print(add_throughput * ns_s); # convert to seconds
246   print("---------------------");
247
```

```
  print("Triad execution time seconds: (avg, min, max):");
  print((as.f64(time_triad_total) / (as.f64(ntimes - 1))) / ns_s);
  print(as.f64(time_triad_min) / ns_s);
  print(as.f64(time_triad_max) / ns_s);
  print("--------------------");

  print("Triad throughput (GB/s):");
  print(triad_throughput * ns_s); # convert to seconds

} else {

  print("STREAM run result is wrong.");
  print("No execution times are displayed.");

}
print("====================");
```

C.2.3 Mandelbrot

Code C.19: Mandelbrot benchmark implementation.

```
# -----------------------------------------------
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
# Modifications Copyright 2022 The DAPHNE Consortium
#
# -----------------------------------------------

# Script to perform Mandelbrot benchmark for DAPHNE
# Following the reference implementation
# https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_openmp/
    mandelbrot_openmp.c


# INPUT PARAMETERS:
# -----------------------------------------------
```

```
# NAME  TYPE    DEFAULT     MEANING
# ------------------------------------------------
# n     integer ---     Pixels resulting in n x n grid
# f_set string  ---     Outputfile name for Mandelbrot
# ------------------------------------------------

# The x and y range is fixed, but the number of grid steps changes with n
# An example of output file wit path: f_set=\"../mandelbrot_set.csv\"

# OUTPUT:
# ------------------------------------------------
# Mandelbrot set (.csv)
# Reports execution time in seconds
# ------------------------------------------------

# ------------------------------------------------
# Global arguments
# ------------------------------------------------
# Width of base pixel grid
width = $n;
# Hight of base pixel grid
height = $n;
# Number of iterations
n_iter = 2000;

# scaling factors for ns to second
ns_s = 1000000000.0;

# set boundary for sequence stability
tol = 2.0;

# borders for real and imaginary axis of grid
x_min = -2.25;
x_max = 1.25;
y_min = -1.75;
y_max = 1.75;


# ------------------------------------------------
# Routine
# ------------------------------------------------

print("==================");
print("Running Mandelbrot");

time_start = now();

# vectors of ones
ones_x = fill(1.0,height,1); # hight x 1
ones_y = fill(1.0,1,width); # 1 x width

# calculate step size given width and height
step_x = (x_max - x_min) / as.f64(width - 1);
step_y = (y_min - y_max) / as.f64(height - 1);
```

```
85
86  # generate seq vecotrs for x and y axis
87  x = seq(x_min,x_max,step_x);
88  y = seq(y_max,y_min,step_y);
89
90  # make real and imaginary part matrices for c
91    # real is just the x-pixel coordinates
92    # each column has the same x-value
93    # imaginary the y-pixel coordinates
94    # each row has the same y-value
95  c_re = ones_x @ t(x); # hi x 1 @ 1 x width
96  c_im = y @ ones_y; # hi x 1 @ 1 x width
97
98  # generate starting z matrices
99  z_re = fill(0.0,height,width);
100 z_im = fill(0.0,height,width);
101 # k matriz for bound check
102 k = fill(0.0,height,width);
103
104 time_initialize = now() - time_start;
105 print("--------------------");
106 print("Initialization time:");
107 print(as.f64(time_initialize) / ns_s);
108 print("--------------------");
109
110 time_intermediate = now();
111
112 for (i in 1:n_iter) {
113
114   # square z
115   # rule: (x + yi)(u + vi) = (xu - yv) + (xv + yu)i
116   z_re_t = z_re * z_re - z_im * z_im;
117   z_im_t = z_re * z_im + z_im * z_re;
118
119   # add c
120   z_re = z_re_t + c_re;
121   z_im = z_im_t + c_im;
122
123   # out of bound check
124   # absolut value is the euklidian norm
125   z_abs = sqrt(z_re * z_re + z_im * z_im);
126   # if not out of bound, add 1 to k
127   k = k + (z_abs <= tol);
128
129 }
130
131 time_loop = now() - time_intermediate;
132 print("Loop time:");
133 print(as.f64(time_loop) / ns_s);
134 print("--------------------");
135
136 # Export matrix with values k, which iteration run out of bound
137 writeMatrix(as.matrix.si64(k),$f_set);
138
```

```
139
140 print("Mandlebrot finished. Mandelbrot set written to specified file.");
141 print("====================");
```

C.2.4 PTRANS

Code C.20: PTRANS benchmark implementation.

```
1  # ----------------------------------------------
2  #
3  # Licensed to the Apache Software Foundation (ASF) under one
4  # or more contributor license agreements.  See the NOTICE file
5  # distributed with this work for additional information
6  # regarding copyright ownership.  The ASF licenses this file
7  # to you under the Apache License, Version 2.0 (the
8  # "License"); you may not use this file except in compliance
9  # with the License.  You may obtain a copy of the License at
10 #
11 #   http://www.apache.org/licenses/LICENSE-2.0
12 #
13 # Unless required by applicable law or agreed to in writing,
14 # software distributed under the License is distributed on an
15 # "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
16 # KIND, either express or implied.  See the License for the
17 # specific language governing permissions and limitations
18 # under the License.
19 #
20 # Modifications Copyright 2022 The DAPHNE Consortium
21 #
22 # ----------------------------------------------
23
24 # Script to perform PTRANS benchmark for DAPHNE
25 # Details on the benchmark: https://icl.utk.edu/hpcc/
26 # Reference Impelmentation:
27 #https://github.com/icl-utk-edu/hpcc/blob/main/PTRANS/pdtrans.c
28
29 # INPUT PARAMETERS:
30 # ----------------------------------------------
31 # NAME      TYPE        DEFAULT     MEANING
32 # ----------------------------------------------
33 # n         Integer     ---       Size of input matrix
34 # ----------------------------------------------
35 #
36 # OUTPUT:
37 # ----------------------------------------------
38 # NAME  TYPE    DEFAULT   MEANING
39 # ----------------------------------------------
40 # Reports execution time
41 # ----------------------------------------------
42
43 # ----------------------------------------------
44 # Global arguments
```

```
45 # ------------------------------------------------
46 # desired size
47 n = $n; # from command line input n=?
48
49 # set tolerance ('epsilon') as in multiLogReg.daphne
50 # example for GitHub DAPHNE repo
51 tol = 0.000001;
52
53 # scaling factors for ns to second
54 ns_s = 1000000000.0;
55
56 # ------------------------------------------------
57 # Routine
58 # ------------------------------------------------
59
60 print("==================");
61 print("Running PTRANS");
62
63 time_start = now();
64
65 # generating matrix
66 A = rand(n,n,0.0,1.0,1,-1);
67 check_sum = 0.0;
68
69 # for checking
70 A_sum = sum(A,1); # col sums of A
71
72 time_initialize = now() - time_start;
73 print("-------------------");
74 print("Initialization time:");
75 print(as.f64(time_initialize) / ns_s);
76 print("-------------------");
77
78 # Transpose
79 time_start = now();
80 B = t(A);
81
82 # Transpose timing
83 time_transpose = now() - time_start;
84
85 # check results with row sums being equal to A column sum
86 B_sum = t(sum(B,0)); # row sum as row vector
87 check = (abs(A_sum - B_sum) > tol); # matrix with zeros if abs() is below tol
88 check_sum = check_sum + sum(check); # add sum matrix to check sum, should be
       zero if correct
89
90
91 if (check_sum == 0.0) {
92
93   print("PTRANS run results are correct.");
94   print("-------------------");
95   print("Transpose execution time:");
96   print(as.f64(time_transpose) / ns_s);
97
```

```
98  } else {
99
100   print("PTRANS run result is wrong.");
101   print("No execution times are displayed.");
102
103 }
104 print("====================");
```