FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D2.2
Programming model and implementation design

WP 2 – A unified analytics programming model

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	5 Feb 2016	H. Vandierendonck, K. Murphy, P. Pratikakis, M. Arif, J. Sun, D.S. Nikolopoulos	Initial Version

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This document describes the current status of the implementation of the ASAP unified programming model. The programming model is used to implement individual steps in a workflow. Programmers who are experts in analytics operations and/or high-performance computing can use this programming model to implement high-performance analytics operators. The majority of users would, however, utilize a higher-level workflow description language as defined in ASAP Deliverable D5.2 [10].

This document describes three contributions: (i) a compiler to translate a workflow described in the workflow description language to the low-level programming language defined in ASAP Deliverable D2.1 [17], or to equivalent Spark code; (ii) a compiler for the low-level programming language; and (iii) a library implementing a few operators to demonstrate the use and efficacy of the language and compiler.

Contents

1 Introduction 7
2 Design 7
2.1 Metadata Components 7
2.2 Compiler Module 8
3 Implementation 12
3.1 Data Structures 13
3.2 Algorithm 13
3.3 Availability 14
4 Low-Level Language (Swan) Compiler 14
5 Evaluation 15
5.1 Setup 15
5.2 Intra-Node Parallelism 16
5.3 Parallel Input 17
5.4 Workflow Fusion 18
5.5 Data Structures 19
5.6 Related Work 20
6 Conclusion 22

List of Tables

1 Repositories holding components of the Swan compiler. 15
2 Data set description. 15

List of Figures

1 Materialized dataset 8
2 Materialized operator 9
3 User Workflow Description for TF/IDF followed by K-means 10
4 Argument declaration rule for TF/IDF and K-means 11
5 Signature rule TF/IDF and TF/IDF-K-means 11
6 Typedef declaration for TF/IDF 12
7 Example template code for input task of TF/IDF 12
8 Class definition for operator 13
9 Class definition for signature 14
10 Self-relative performance scalability of the K-Means operator. 16
11 Self-relative parallel scalability of the TF/IDF operator. 18
12 Impact of merging operators on execution time 19
13 Impact of container type on the TF/IDF–Kmeans workflow on execution time . . . 19

1 Introduction

ASAP aims to make it easy for users to describe and run data analytics queries by selecting the most suitable and efficient execution engines. There should be no restriction on the type of execution engine or on the format and medium used for input and output datasets.

ASAP provides a workflow tool, as described in ASAP Deliverable D5.2 [10], to facilitate the visual creation and adaptation of data analytics queries enabling analytics experts to create and change queries by dragging operator boxes and dataset flow arrows in a graphical interface into a suitable pictorial representation of a query workflow.

Subsequent to query creation in the workflow tool, a multi-engine resource scheduling platform, IReS [6] takes the workflow description of the analytics query and co-operates with a set of runtimes and data stores in order to effect the best implementation and execution of the query.

A bridge is needed between the high level abstract description of a workflow query and the actual provisioning and execution of codes that can run the query. This bridge is satisfied by the ASAP compiler for WP2. It maps the user's high level workflow query, as created within the workflow tool [10], to codes for the low-level programming language Swan [29] using metadata which describes how Swan operators are materialized. ASAP's scheduler can subsequently use these codes to profile and select optimum runtime engines and data stores.

2 Design

2.1 Metadata Components

The compilation process, from workflow to execution, requires metadata descriptions for materialized operators and the user's workflow or analytics query description. It indirectly requires metadata descriptions for abstract operators.

Abstract operators This is not directly used by the compiler itself but is included here for context. It provides the workflow tool with information on available operators, inputs and outputs so these can be presented as options to the data analyst when creating their query at the graphical interface. The information is limited to what the data analyst needs at the point of workflow creation. A matching process between abstract and materialized operators allows the scheduler to identify and profile all options for how an abstract operator may be materialized.

Materialized operators This provides information to the IReS scheduler so it knows how to materialise operators as described in [6]. The description is more detailed than that provided by abstract operator descriptions and typically includes:

1. Input/output formats
2. File formats

```
"operators": [{
        "name": "textDirectoryDataset",
        "cost": "0.00",
        "status": "stopped",
        "type": "dataset",
        "description": "textDirectoryDataset",
        "Constraints": {
            "DataInfo": {
                "type": "freetext"
            },
            "Engine": {
                "FS": "Standard"
            },
            "type": "directory"
        },
        "Execution": {
            "path": "/var/shared/projects/asap/inputs/operators/ tfidf_input /"
        },
        "input": []
},
```

Figure 1: Materialized dataset

3. Execution engines
4. Argument requirements
5. Default argument values
6. Structure of variable declarations
7. Typedefs options for algorithm data structures

By design we permit alternative descriptions for some operator properties, for example typedef definitions. This makes it possible to profile execution performance using different algorithm properties in different scenarios. For example we may find that a list data structure is better for some particular datasets but not others. An example of a materialized dataset is shown in Figure 1. An example of a materialized operator for TF/IDF is shown in Figure 2.

User's workflow This is a representation of what the user created in the workflow tool provided by ASAP Deliverable D5.2 [10]. It records the sequence or graph of operators and interconnecting data flows occurring within the query, and is used by the compiler to drive the generation of Swan code. A workflow description is shown in Figure 3.

2.2 Compiler Module

The compiler reads and stores:

```
{
        "name": " tfidf_cilk_map ",
        "cost": "0.00",
        "status": "stopped",
        "type": "operator",
        "description": "TFIDF",
        "Constraints": {
                "EngineSpecification": {
                        "FS": "standard"
                },
                "type": "directory",
                "runFile": " tfidf_cilk_map ",
                "Algorithm": {
                        "name": " tfidf ",
                        "dstruct_type": "word_map_type"
                },
                "Input" : {
                        "Engine": {
                                "FS": "standard"
                        },
                        "number": "1",
                        "type": "textDirectoryDataset"
                },
                "Output" : {
                        "Engine": {
                                "FS": "standard"
                        },
                        "number": "1",
                        "type": "arffDataset"
                }
        }
},
```

Figure 2: Materialized operator

```
"workflow": {
    "nodes": [{
            "id": "1",
            "taskids": ["1"],
            "name": "tfidf"
    }, {
            "id": "2",
            "taskids": ["2"],
            "name": "kmeans"
    }],
    "edges": [{
            "sourceId": "1",
            "targetId": "2"
    }],
    "taskLinks": [ ],
    "tasks": [{
                "id": "1",
                "nodeId": "1",
                "name": "tfidf",
                "operator": {
                        "constraints": {
                                "input": "1",
                                "input0": "tfidf_input",
                                "output": "1",
                                "output0": "tfidf_output.arff",
                                "opSpecification": {
                                        "algorithm": "tfidf_map",
                                        "args": {"num_clusters": "4",
                                                "max_iters": "5",
                                                "force_dense": "true"}

        }       }       }       }, {
                "id": "2",
                "nodeId": "2",
                "name": "kmeans",
                "operator": {
                        "constraints": {
                                "input": "1",
                                "input0": "tfidf_output.arff",
                                "output": "1",
                                "output0": "kmeans_output.txt",
                                "opSpecification": {
                                        "algorithm": "kmeans",
                                        "args": {"num_clusters": "4",
                                                "max_iters": "5",
                                                "force_dense": "true"}
    }       }       }       }]
```

Figure 3: User Workflow Description for TF/IDF followed by K-means

```
{
        "type": "arg_declaration",
        "algorithm.names": ["tfidf_and_kmeans"],
        "argTemplates": [{"max_iters": "const␣int␣VARIN␣=␣VAROUT;", "num_clusters": "const␣int
            ␣VARIN␣=␣VAROUT;", "force_dense": "const␣bool␣VARIN␣=␣VAROUT;", "by_words": "
            const␣bool␣VARIN␣=␣VAROUT;", "do_sort": "const␣bool␣VARIN␣=␣VAROUT;", "
            rnd_init": "unsigned␣int␣VARIN␣=␣VAROUT;"}],
        "argDefaults": [{"max_iters": "0", "num_clusters": "8", "force_dense": "false", "
            by_words": "false", "do_sort": "false", " rnd_init ": "1"}]
},
```

Figure 4: Argument declaration rule for TF/IDF and K-means

```
{
        "type": "signature_rule",
        "algorithm.names": [" tfidf ", "tfidf_and_kmeans"],
        "input": " get_dir_listing (FILE_PARAM1,␣dir_list);",
        "output": "output(DATA_PARAM1,␣FILE_PARAM1);",
        "run": " tfidf (DATA_PARAM1,␣OP_PARAM1,␣OP_PARAM2);"
},
```

Figure 5: Signature rule TF/IDF and TF/IDF-K-means

1. The metadata for materialized operators
2. The user's work-flow description

and generates Swan source code [29] which can execute the user's analytics query.

Phase 1 of the compiler loads descriptions of operators and workflows from the metadata files into instances of descriptor classes which map operators from the user's workflow to source code and subsequently executable code. Figures 4, 5 and 6 show examples of rules (read from the materialized operators file) for argument declarations, signature and typedef rules for certain operators, including TF/IDF.

Once this is loaded the compiler can infer rules around the use of individual operators. For example, it will be able to build the function signature for calls to core functions such input, output and operator run statements. It can work out what type definitions are required for different operators, what arguments are permitted to operators such as K-means and what the default values are if these are not supplied by the user.

Phase 2 The compiler generates the code by loading skeletal source code from template files. These contain sections of Swan source code calling into a library with pre-defined functions. The skeletal code moreover contains placeholders requiring substitution with actual values

```
{
        "type": "typedef",
        "algorithm.names": [" tfidf "],
        "algorithm.types": ["word_map_type"],
        "types": ["typedef asap::word_list<std::deque<const char*>, asap::
            word_bank_managed> directory_listing_type;",
                "typedef asap::word_map<std::map<const char *, size_t, asap::text::charp_cmp
                    >, asap::word_bank_pre_alloc> word_map_type;",
                "typedef asap::word_list<std::vector<std::pair<const char * const, size_t>>,
                    asap::word_bank_pre_alloc> word_list_type;",
                "typedef asap::sparse_vector<size_t, float , false , asap::
                    mm_no_ownership_policy> vector_type;",
                "typedef asap::word_map<std::map<const char *, asap::appear_count<size_t,
                    typename vector_type::index_type>, asap::text::charp_cmp>, asap::
                    word_bank_pre_alloc> word_map_type2;",
                "typedef asap::data_set<vector_type, word_map_type2, directory_listing_type>
                    data_set_type;"
        ]
},
```

Figure 6: Typedef declaration for TF/IDF

```
// Directory listing
get_time( begin );

 directory_listing_type    dir_list ;
asap:: get_directory_listing ( FILE_PARAM1, dir_list );
get_time (end);
print_time ("directory  listing ", begin, end);
```

Figure 7: Example template code for input task of TF/IDF

and/or code sections. The compiler reads the templates and performs these substitions using operator rules to produce operator codes. An example of one such template code file which provides template code for the input section for the TF/IDF operator is shown in (Figure 7). Here FILE_PARAM1 will be replaced by user supplied information before code generation. Additionally the compiler wholly generates code sections for input, output and argument declarations and initializations

3 Implementation

Python [1] was chosen as the development language for the compiler due to its simplistic syntax and dynamic typed features which make it flexible and scalable. JSON was chosen as the first

```
""" holds data about operator constraints """
class OperatorConstraint:
    def __init__(self, fs, runFile, algname, alg_dstructtype, inputs, outputs):
        self.EngineSpecification_FS = fs
        self.runFile = runFile
        self.algname = algname
        self.algtype = alg_dstructtype
        self.inputConstraint = []
        self.outputConstraint = []
        for i in range(0,int(inputs["number"])):
            inputConstraint = IOConstraint(inputs["Engine"]["FS"],
                                           inputs["type"])
            self.inputConstraint.append(inputConstraint)
        for i in range(0,int(outputs["number"])):
            outputConstraint = IOConstraint(outputs["Engine"]["FS"],
                                           outputs["type"])
            self.outputConstraint.append(outputConstraint)

""" holds data about operators """
class Operator:

    def __init__(self, name, description, constraint, inlist, status="stopped"):
        self.name = name
        self.description = description
        self.constraint = constraint
        self.status = status
```

Figure 8: Class definition for operator

supported metadata language of choice mainly to facilitate early integration with the workflow tool [10] and scheduler [6].

3.1 Data Structures

Python classes have been defined for representing operators, operator rules and datasets. Their separate representations allow for a loosely coupled approach where datasets and language construct rules can equally be applied to any or many operators in a workflow schema. Figures 8 and 9 show examples of class definitions for an operator and a signature rule.

Separate instances are created for each occurrence in a workflow and attached to related or enclosing operator instances.

3.2 Algorithm

The compiler algorithm is driven by:

```
class Signature:
    def __init__(self, input, output, run):
        self.input = input
        self.output = output
        self.run = run
```

Figure 9: Class definition for signature

1. The flow of operators in the user's workflow description.

2. The general sequencing and layout of a Swan/C++ program.

Initially the compiler parses metadata from the materialized operators library and creates a hierarchy of descriptor instances which represent operator and language construct rules. These may be anything from operator objects to rules for declarations, typedefs and arguments associated with operators. Code generation proceeds by:

1. Parsing and looping around the user's workflow of operators.

   (a) Referencing the descriptor instances.

   (b) Referencing template code files and substituting placeholders.

   (c) Printing the code section.

Typically declarations for input, output and operator arguments are generated directly and wholly from the materialized operator rules, replacing variable names with internally generated compiler ones and filenames with those supplied by the user.

3.3 Availability

The source code for the workflow compiler and the library of operator implementations in Swan are available online from `github.com/hvdieren/asap_operators/`. The workflow compiler has been tested on four example workflows. The operators implemented in the library are K-means clustering, word count, term frequency/inverse document frequency calculation for a set of documents, numeric dataset input/output in WEKA's Attribute-Relation File Format (ARFF) format [8] and a directory listing operator.

4 Low-Level Language (Swan) Compiler

One of the goals of the ASAP Work Package 2 is to implement code transformations for the Swan task dataflow programming language [29]. These code transformations are intended to restructure code in order to improve performance. As a first step towards this goal, we have extended the

Table 1: Repositories holding components of the Swan compiler.

Component	Repository
LLVM	`https://github.com/hvdieren/swan_llvm`
Clang	`https://github.com/hvdieren/swan_clang`
Runtime	`https://github.com/hvdieren/swan_runtime`
compiler-rt	`https://github.com/llvm-mirror/compiler-rt` version b6967623458e90b89b4e8a5311c5c9e0758bcb6e
tests	`https://github.com/hvdieren/swan_tests`

Table 2: Data set description.

Input	Documents	Bytes	Distinct words
Mix	23432	62.8 MB	184743
NSF Abstracts	101483	310.9 MB	267914
Gutenberg	52361	19.4 GB	7614691

Clang [12] compiler and defined a front-end parser that recognizes the language extensions defined by Swan. These include, primarily, the addition of dataflow dependences to tasks in the Cilk language [7] and versioned objects that represent the data that flows between tasks. These constructs are recognized by the Clang compiler and they are correctly translated to common operations in the LLVM Intermediate Representation (which is clang's target language) and also include runtime library calls. The runtime library is an extension of the open source Intel Cilkplus library.

The modified versions of the software tools are available on github as indicated in Table 1. The tests repository holds specific instructions to setup the Swan compiler. It also presents a few test cases to check that the compiler is working correctly.

5 Evaluation

The following evaluation section has been published at the First International Workshop on Multi-Engine Data Analytics (MEDAL) [28]. We have provided additional evaluation on a larger data set, namely a 20 GB collection of books from the Gutenberg project.

5.1 Setup

We start our investigation at a small scale, focusing on the activities on a single node as these allow us to better understand the performance of operators and workflows.

Performing analytics on a single node is important as a single-node can be built with a large amount of working memory (up to 16 TB) and many processing cores (over a 100). Such a system

could efficiently process many real-world data sets. However, we expect that our conclusions remain valid when applied to scale-out systems, as optimizing the performance of nodes in isolation is crucial to optimize the system overall.

To test the importance of the identified optimizations, we implement two analytics operators in the Cilkplus extension of C++, a programming language designed for high-performance and parallel computing at MIT, first developed over two decades ago and continuously refined since then. Cilkplus, now commercialized by Intel, supports the construction of parallel tasks through language constructs that express parallelism and vectorization (SIMDization) in an easily accessible way. In the Cilkplus model, each thread of computation is bound to a processing core. The principles utilized should apply to other languages and parallel constructs, e.g., Java streams.

We study two operators: term frequency–inverse document frequency (TF/IDF) and K-means clustering. TF/IDF extracts words from text documents and rates the importance of a word on the basis of its frequency of occurrence within a specific document as well as within the whole set of documents. K-means clustering is an unsupervised classification technique that allows for the grouping of similar data items described as numeric vectors.

We evaluate the operators on three data sets: a collection of short news articles from Reuters, the NSF Abstracts dataset and a collection of books from the Gutenberg project.

5.2 Intra-Node Parallelism

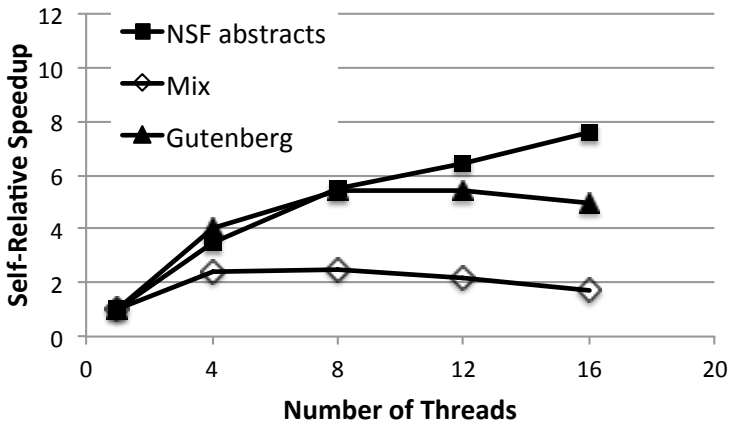

Figure 10: Self-relative performance scalability of the K-Means operator.

Many problems in data mining are trivially compute-bound, especially learning algorithms using neural networks, support vector machines and the like, which utilize computationally demanding hyperbolic functions and can require many iterations to train the model. It should go without saying that algorithms like these can be accelerated using high degrees of intra-node parallelism.

K-means clustering is perhaps one of the cheapest unsupervised learning algorithms. As such, we will use K-means clustering to demonstrate that data analytics operations benefit from intra-node parallelism. Figure 10 shows the self-relative speedup of the K-means clustering algorithm

on our three datasets (Table 2). We use the algorithm to assign documents to one of 8 clusters based on their normalized TF/IDF scores.

The self-relative speedup shows how much performance is improved by utilizing multiple CPU cores. The speedup obtained is sensitive to the data set operated on: The *NSF Abstracts* data set has about 100,000 documents and is sped up nearly 8 times using intra-node parallelism. The *Mix* data set has around 23,000 documents, which is sufficient only for a 2.5 x speedup. The *Gutenberg* data set has double the number of documents of *Mix* and achieves twice the speedup. This effect is explained by the parallel loops in K-means clustering, which are all loops iterating over the documents. As the number of documents grows, so does the parallel scalability.

The execution time of our implementation is furthermore short in comparison to other implementations. We compared the execution time of our K-means clustering implementation against WEKA [8] (version 3.6.13). Using the "SimpleKMeans" algorithm, a single-threaded K-Means algorithm, on the same data sets requires over 2 hours, after which we aborted the execution. In contrast, executing our implementation sequentially required 3.3s and 40.9s for the *Mix* and *NSF Abstracts* data sets respectively. Note that while we did not see the execution of WEKA through to the end, we have verified that our WEKA installation works correctly on small data sets.

While our implementation is significantly faster than WEKA, this is not automatic. Several key optimizations were required to achieve the performance of our algorithm: (i) Using sparse vectors to represent inherently sparse data. (ii) Re-cycling data structures throughout the K-means iterations to avoid redundant data copies and memory pressure. E.g., we do not create new objects during the iterations of the K-means algorithm.

The conclusion of this experiment is thus that (i) intra-node parallelism is an important opportunity to accelerate data analytics, especially on larger data sets; (ii) the implementation and the choice of data structures has a huge influence on execution time; (iii) parallelism can be exploited without casting the algorithms in map/reduce form.

5.3 Parallel Input

A code that is well-optimized and where CPU is a bottleneck can also benefit from parallelizing I/O operations. Under these circumstances, CPU utilization is high and I/O resources are underutilized, including local disk and network resources. Intra-node parallelism can thus increase the utilization of disk and network resources.

In this section we study the problem of calculating the term frequency–inverse document frequency (TF/IDF) [24] property of a set of documents. Our implementation collects term frequencies (word counts) for each of the documents in the set. Moreover, a list of all unique terms across the documents is constructed. This list is annotated with the number of documents where the word occurs. In a first phase, the per-document term frequencies and the overall term-document count properties are collected using dedicated hash tables, mapping a word to a term frequency or an overall document count. In a second phase, we calculate for each document the per-term TF/IDF score using the hash tables described above. For each document, a sparse TF/IDF vector

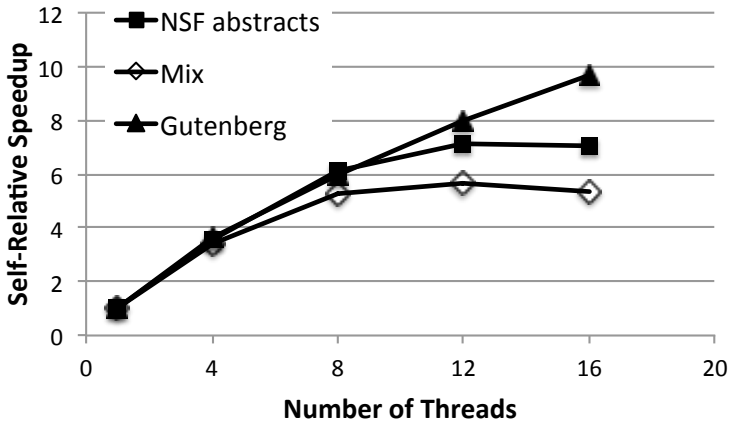

Figure 11: Self-relative parallel scalability of the TF/IDF operator.

is constructed, sorted by term IDs and written to the output file in Attribute-Relation File Format (ARFF) format [8]. The first phase can be executed in parallel for each of the documents. The main limitation to obtain speedup here is bandwidth to the storage system. The second phase is not parallelized as the ARFF format does not facilitate parallel output.

While the TF/IDF problem is mainly concerned with data input, tokenization and hash table operations, it benefits strongly from intra-node parallelism (Figure 11). It speeds up by nearly 6-fold for the *Mix* data set and by 7-fold for the *NSF Abstracts* data set. The *Gutenberg* data set, which is substantially larger, speeds up nearly 10-fold. Parallelizing output is important as well. However, file formats are often designed in such a way that parallel I/O becomes hard.

5.4 Workflow Fusion

As pointed out above, I/O is both costly and hard to parallelize. As such, avoiding I/O is always a good optimization. Figure 12 shows the execution time of the TF/IDF–K-Means workflow when executing the TF/IDF and K-Means operators as discrete operators that communicate by storing the intermediate TF/IDF scores on disk, versus a merged operator without storage of the intermediates. The results clearly demonstrate that dumping data to disk has a high latency. In this experiment, the data is dumped to a local hard disk. Both the output of the TF/IDF scores and the subsequent input are executed by a single thread because the file format utilized (ARFF [8]) does not easily support parallel I/O. In contrast, transforming the data when it is stored in-memory is much faster and parallelizes well.

The presence of intra-node parallelism is an important differentiator as to whether I/O bears much overhead or not. On a single-threaded execution, I/O increases execution time by 36.9%. On 16 threads, however, I/O makes the execution 3.84 times slower because it does not parallelize.

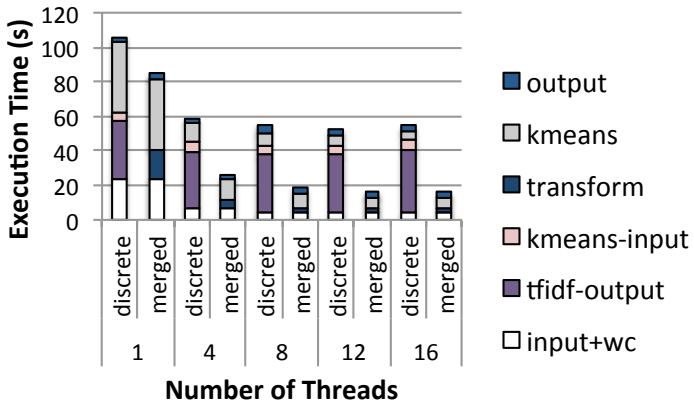

Figure 12: Execution time of the TF/IDF–K-Means workflow when executing the TF/IDF and K-Means operators as discrete steps communicating through file I/O, versus a merged operator with storage of the TF/IDF scores. Uses the *NSF Abstracts* input.

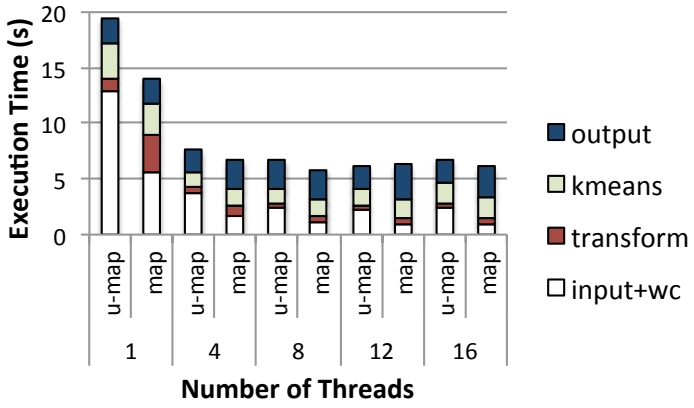

Figure 13: Execution time of the TF/IDF–K-Means workflow on the *Mix* input using a `std::unordered_map` (u-map) or a `std::map`.

5.5 Data Structures

Algorithms use data structures to store input, output and internal data sets, The choice of these data structures impact performance. In the case of TF/IDF, the key data structures are the dictionaries storing unique words and their frequencies. Figure 13 shows the execution time of TF/IDF–K-Means workflow on the *Mix* data set and a varying number of threads. Results for the largest *NSF Abstracts* data set are more dramatic.

The results demonstrate that the input and word-count step ("input+wc" in Figure 13) is faster when using the `std::map` data structure as opposed to the `std::unordered_map` data structure. The first is implemented as a red-black tree, while the latter is implemented as a hash table.

Moreover, the unordered map is pre-sized to hold 4K items to minimize resizing overhead.

While reading documents and counting words is faster with a map, the subsequent data transformation step is slower using a map, especially on one thread. This follows as the input and word-count phase is write-intensive, consisting of frequent insertion of values in the dictionary. Insertion in the unordered map (a hash table) is inefficient due to (i) resize operations, which requires re-hashing all elements, (ii) memory pressure, as the array underlying the hash table is by construction both sparse (to approximate $O(1)$ operations) and very large (due to the data sets used). In contrast, the transformation step performs only lookups on the hash table, which are known to be faster on the unordered map $O(1)$ as opposed to the map $O(\log n)$.

However, the transformation step scales much better with an increasing number of threads when using the map: it scales to 6.1 x on 16 threads using the map, while it scales only to 3.4 x using the unordered map data structure. This is in part due to the memory consumption. In particular, using the *Mix* data set, main memory consumption is 420 MB with the map, while it rises to 12.8 GB using the unordered map.

Likewise, the output phase performs lookups only on the dictionaries and thus favours the unordered map. Moreover, the output phase is hard to parallelize.

We conclude that selection of the internal data structures has a significant impact on execution time. Moreover, different steps of a workflow may execute faster using different data structures. As such, the choice of internal data structure must be taken judiciously, depending on the overall time taken by each step of the workflow and also on the extent to which each phase can be parallelized.

5.6 Related Work

The performance of data analytics frameworks is an important concern. Various studies have been performed to probe into the efficiency of distributed data analytics frameworks such as Hadoop and Spark. It is known that generalized data management systems such as structured databases incur a performance penalty through generalization. Such a result can be fairly extrapolated to unstructured big data stores and processing frameworks. These studies, however, indicate deep performance issues that transcend "the cost of generalization."

Pavlo *et al* compare map/reduce systems against distributed DBMSes [21]. They observe that the basic control flow of map/reduce has existed in parallel SQL database management systems (DBMS) for over 20 years. They compare the map/reduce and parallel SQL paradigms and find interesting trade-offs in performance between these approaches. They find that importing data in the DBMS takes substantial time and configuration of one of the tested databases and required repeated vendor assistance. The DBMS, however, proved 2.3X faster than map/reduce on 100 nodes. They conclude that map/reduce is less efficient and attribute this to its design, in particular the lack of an index over the data.

Ousterhout *et al* analyse real-life peta-scale workloads executing over Spark [20]. They find that CPU is more often a bottleneck than I/O and that network performance has little impact on job completion time. Moreover, they find that straggler nodes can be identified and that in most cases

the cause for straggling can be identified. The authors also note that their analysis is a snapshot in time. They expect that the results of their analysis will change as data analytics systems evolve. As such, the performance bottlenecks will also change.

Han *et al* perform a similar analysis for graph analytics frameworks [9]. They identified common optimizations and framework-specific optimizations, identifying which brought most performance improvement. They also identified potential areas for improvement, e.g., load balancing and adjacency list data structures that are memory and mutation-efficient (Giraph) and reducing communication overheads in GraphLab's asynchronous mode. Satish *et al* [25] similarly study graph analytics frameworks. Their study proposes hand-optimised implementations of the graph analytics operations and uses these to identify important bottlenecks in the frameworks. They find that their hand-optimized codes can outperform programmer-friendly frameworks by up to 560-fold. More importantly, they identify that GraphLab, Giraph and Socialite have poor network utilization. This is in stark contrast with Spark and map/reduce which have been reported as CPU-limited.

McSherry, Isard and Murray propose a different way to assess the efficiency of distributed data analytics frameworks [16]. The "COST" metric determines the "Configuration that Outperforms a Single Thread." The idea is to compare the performance of a workload executing on a distributed data analytics framework against an implementation of the algorithm executing within a conventional programming language. In their case, the reference implementation was single-threaded C# code. They apply the COST metric to a number of applications and data analytics frameworks. For instance, they find that Naiad [18] has a COST of 16 cores for calculating PageRank on the Twitter graph, while GraphLab [13] has a COST of 512 cores. GraphX [30] has an unbounded COST, i.e., it is never as fast as the single-threaded implementation no matter how many cores and nodes are used. The authors point out that such high overheads simplify creating *scalable systems*, i.e., it is much easier to scale an inefficient software system over a large cluster than it is to scale an efficient software system.

Several authors have investigated analytics frameworks for shared-memory systems. The argument for data analytics on single nodes is based on (i) the possibility of building nodes with up to 16 TB of working (DRAM) memory and (ii) the higher efficiency and simplicity of developing shared memory software systems as opposed to distributed memory system solutions. The latter argument is especially important for graph processing workloads which are extremely communication- and synchronization-intensive. As such, Shun and Blelloch [26] have developed Ligra, a light-weight graph analytics system for shared memory systems that scales well to large core counts. For graphs that do not fit in working memory at once, Kyrola *et al* have designed GraphChi [11], a graph analytics framework for graphs stored on disk. GraphChi partitions graphs in shards. It then reads shards of the graph in memory, performs calculations on the shard, applies the changes to disk and then moves on to the next graph. GraphChi is efficient due to its partitioning technique which retains good and predictable locality of edge destinations within each shard. Zhang *et al* take the Ligra approach further and optimize graph analytics for non-uniform memory architectures (NUMA) [32]. NUMA systems are partitioned and each CPU socket has faster access to its associated partition of the memory than to other partitions. They use the same

graph partitioning technique as GraphChi to partition the graph across NUMA partitions and to bias main memory accesses to the local NUMA memory.

Research has also been invested in map/reduce frameworks [5] for shared memory systems [23, 31, 27]. Mao *et al* have proposed optimizations to Phoenix [15]. In particular, they point out the need for NUMA-aware memory allocation. Chen *et al* optimize Phoenix by tiling, a data locality optimization technique [3]. Several authors have proposed map/reduce systems to program accelerators, including the Sony/Toshiba/IBM's Cell Broadband Engine [4, 22] and Intel's Xeon Phi [14]. On a different note, Arif *et al* have evaluated how standard parallel programming languages weigh up to parallelizing map/reduce applications [2]. In particular, they have found that OpenMP [19] has insufficient support for reducing container data structures, e.g., arrays and key-value maps. OpenMP focusses its reduction operations on scalar values.

6 Conclusion

The deliverable describes the first version of the ASAP compiler which transforms a workflow description in high level form to 0 codes for execution on heterogeneous engines. In the design attention was given to ensure the compiler would be flexible and extensible and facilitate integration with interfacing components [10] and [6] by using compatible metadata schemas.

The compiler has been tested on a series of simple workflows including, TF/IDF, k-means and a combined in-memory TF/IDF-K-means operation. These have shown successful mappings between workflows and executions in the Swan execution environment. Profiling shows efficiency gains in selecting the in-memory tfidf-k-means workflow. Current work is underway to install Swan/C++ operator codes within the data centers in the ASAP framework so we can re-create the workflow executed by the Spark engine, and execute it with Swan codes. This will provide further validation of the compiler as a transformation from analytics workflows to heterogeneous executions environments.

Bibliography

[1] https://www.python.org/.

[2] M. Arif and H. Vandierendonck. A case study of OpenMP applied to map/reduce-style computations. In *Intl. Workshop on OpenMP*, page 6, October 2015.

[3] R. Chen and H. Chen. Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM Trans. Archit. Code Optim.*, 10(1):3:1–3:30, April 2013.

[4] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, Sept 2009.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the 6th Symp. on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, 2004.

[6] K. Doka, N. Papailiou, C. Mantas, V. Giannakouris, and D. Tsoumakos. ASAP deliverable 3.2: IReS platform, 2015.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. of the 1998 ACM SIGPLAN Conf. on Programming language design and implementation*, pages 212–223, 1998.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[9] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014.

[10] V. Kantere and M. Filatov. ASAP deliverable 5.2: Workflow management tool, 2015.

[11] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[12] C. Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[14] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R.S.M. Goh, and R. Huynh. Optimizing the mapreduce framework on Intel Xeon Phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, pages 125–130, Oct 2013.

[15] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT Computer Science and Artificial Intelligence Laboratory, 2010.

[16] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[17] K. Murphy, J. Sun, and H. Vandierendonck. ASAP deliverable 2.1: Preliminary definition of ASAP programming model, 2015.

[18] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[19] OpenMP application programming interface, version 4.0. http://www.openmp.org/, July 2013.

[20] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proc. of the 12th USENIX Conf. on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, 2015.

[21] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '09, pages 165–178, 2009.

[22] M.M. Rafique, B. Rose, A.R. Butt, and D.S. Nikolopoulos. CellMR: A framework for supporting MapReduce on asymmetric Cell-based clusters. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[24] G. Salton and M. J. McGill, editors. *Introduction to Modern Information Retrieval*. Mcgraw-Hill, 1983.

[25] N. Satish, N. Sundaram, Md. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 979–990, 2014.

[26] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[27] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[28] H. Vandierendonck, K. L. Murphy, M. Arif, J. Sun, and D. S. Nikolopoulos. Operator and workflow optimization for high-performance analytics. In *Proceedings of the First International Workshop on Multi-Engine Data Analytics (MEDAL)*, volume 1558 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[29] H. Vandierendonck, G. Tzenakis, and D. S Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 1–11. IEEE, 2011.

[30] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First Intl. Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, 2013.

[31] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[32] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *Proc. of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, 2015.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D2.2
Programming model and implementation design

WP 2 – A Unified Analytics Programming Model

Nature: Report

Dissemination: Public