Serverless Big Data Processing using Matrix Multiplication as Example

Sebastian Werner
ISE, TU Berlin, Germany

Jörn Kuhlenkamp
ISE, TU Berlin, Germany

Markus Klems
WeAdvise AG, Munich Germany

Johannes Müller
TU Berlin, Germany

Stefan Tai
ISE, TU Berlin, Germany

***Abstract*—Serverless computing, or Function-as-a-Service (FaaS), is emerging as a popular alternative model to on-demand cloud computing. Function services are executed by a FaaS provider; a client no longer uses cloud infrastructure directly as in traditional cloud consumption. Is serverless computing a feasible and beneficial approach to big data processing, regarding performance, scalability, and cost effectiveness? In this paper, we explore this research question using matrix multiplication as example. We define requirements for the design of serverless big data applications, present a prototype for matrix multiplication using FaaS, and discuss and synthesize insights from results of extensive experimentation. We show that serverless big data processing can lower operational and infrastructure costs without compromising system qualities; serverless computing can even outperform cluster-based distributed compute frameworks regarding performance and scalability.**

***Keywords*-serverless, big data, cloud, matrix multiplication**

I. Introduction

Over the last years, performance and scalability needs for big data processing have been rather successfully addressed. This has been achieved by infrastructure platforms based on distributed computing frameworks[1][2][3][4] that run on servers provisioned by cloud infrastructure services, e.g., AWS EC2. However, operating staff costs and infrastructure costs present significant cost factors for processing data-at-scale. Furthermore, processing big data on cloud infrastructure requires extensive knowledge to define and execute jobs and to deploy, configure, and maintain the required infrastructure platform for running jobs. Therefore, the reduction of both costs and entry barriers for processing data-at-scale have been identified as a grand challenge of data management [1].

For cloud-native applications, serverless computing has emerged as a new paradigm [2], [3]. Serverless computing, or Function-as-a-Service (FaaS), makes infrastructure components transparent to application developers. Instead of using cloud infrastructure directly, developers provide short-running code for functions to be executed and scaled as needed by FaaS providers.

While first prototypical implementations for serverless big data processing have been proposed[5], developers and decision makers require hard evidence to make knowledgeable decisions about using traditional cloud infrastructure versus using FaaS. With our work, we address the research question of whether serverless platforms are feasible and beneficial for analyzing data-at-scale. We build on our extensive expertise and previous work on quality-driven design and evaluation of cloud-based systems [4]–[8] and answer the research question by means of experimental evaluation. We chose distributed matrix multiplication as an example for our analysis due to its relevance in multiple application contexts, including the ongoing research projects DITAS[6] and BloGPV.Blossom[7].

In this paper, we present three contributions over the state-of-the-art:

1) We define generic requirements for guiding the design of serverless big data applications.
2) Based on the requirements, we present the design and corresponding running prototype for distributed matrix multiplication.
3) We perform an extensive experiment-based evaluation of this prototype, discuss and synthesize generic insights for the design of future serverless big data applications.

The remainder of this paper is structured as follows. First, we define generic requirements of serverless big data applications (section II). Second, we present the design (section III) and implementation (section IV) of our prototype. Next in section V, we present the evaluation of our prototype followed by a discussion of related work (section VI) and a conclusion (section VII).

II. System Requirements

We define generic non-functional requirements for guiding the design of serverless big data applications. These include established system requirements, e.g., scalability and performance, as well as new requirements to achieve the overall goals of reducing cost and entry-barriers of

[1]http://spark.apache.org/
[2]http://hadoop.apache.org/
[3]http://flink.apache.org/
[4]http://www.tensorflow.org/

[5]https://github.com/awslabs/lambda-refarch-mapreduce/
[6]https://www.ditas-project.eu/
[7]https://www.ise.tu-berlin.de/menue/projekte/blogpv/

processing data-at-scale. Cost reduction should target both infrastructure costs and operational costs, e.g., the expenditure of time of experts. At the same time, a reduction of entry barriers can be achieved by reducing the required skillset for both infrastructure and processing platform.

To reduce infrastructure costs, appropriate infrastructure platforms must be selected. For cloud infrastructure services, this implies a cost model that bills resource usage in short time intervals for small increments of resources, and provisions resources fast.

A reduction of operational costs can be achieved by reducing the time experts need to understand and operate a solution stack to match application requirements at hand. This can be achieved by using fully managed infrastructure services and by providing a small number of high-level tuning-knobs with well-documented impacts on application qualities. Thus, both requirements benefit the goal of reducing entry barriers.

Therefore, the requirements for our example of matrix multiplication are as follows:

(I) Scalability: More compute resources allow the system to multiply (i) larger matrices effectively and (ii) given matrices faster.

(II) Fully Managed: The system makes use of fully managed cloud services, and, thus, does not require any operation from an application developer (NoOps).

(III) Performance: The system provides better or similar performance, i.e., time-to-solution, as VM-based deployments of distributed compute frameworks.

(IV) Infrastructure Costs: The deployment costs, i.e., aggregated costs for provisioned cloud infrastructure of the system are lower or similar compared to VM-based deployments of distributed computing frameworks.

(V) Tunable: Horizontally scalable distributed systems can add or remove cloud infrastructure resources in a deployment to (i) increase performance and infrastructure cost or (ii) decrease performance and infrastructure cost, respectively. The system allows an application developer to configure this trade-off according to application requirements at hand.

III. System Design

We chose matrix multiplication as an example to showcase how to address the defined system requirements. There are several approaches to distributed matrix multiplication; the challenge is to adapt these approaches to the serverless model, where state transfer between workers is limited, and to design the application to be tuneable in terms of cost and performance.

Our system design consists of three main elements: calculation, storage, and orchestration. Calculation is always handled through stateless functions, as is required by the serverless paradigm. For matrix multiplication, we need three different functions to create the overall solution. These functions get input and store there outputs in the cloud managed storage. Other than operational meta-data, no data is transferred directly between function instances. Orchestration ensures that the workload is distributed and split over the three different function types.

Each computation goes through the same three steps:

1) load required partitions of factors or intermediate results from storage
2) perform designated calculation (multiplication or addition)
3) write result(s) back to storage.

This overall design ensures that the application is compatible with the serverless infrastructure.

A. Computation

Figure 1. Architecture overview showing units of multiplication and accumulating function instances.

Due to resource and time limitations of function instances, first, a basic fixed function architecture is established. This architecture can perform square matrix multiplication, distributed across many functions. Here we selected Strassen's algorithm to ease the distributed calculation.

In order to increase the range of matrices supported and to allow for even larger computations, this basic unit of multiplication can be replicated multiple times and executed in parallel. Additional instances of a different function-type will then accumulate and consolidate partial results as shown in figure 1. Other than with distributed computing platforms, a serverless implementation needs to coordinate these parallel running jobs implicitly.

Computation of partial multiplications can also be sped up by increasing the degree of parallelization combined with a hierarchical accumulation of intermediate results. The degree of parallelization can be adapted in order to tune the cost-speed trade-off for any workload size. A higher degree of parallelization will result in faster overall computation but comes at an increased price.

1) Strassen Compute Unit (SCU): As stated above, we use the Strassen's algorithm for multiplication. Its properties allow distributing one multiplication onto seven independently running lambda functions. They each read the input matrix from storage and write back the result. Since addition is a reasonably cheap operation, a single instance will consolidate all intermediate results and write them back to storage.

The collector stage is initiated after all intermediate values have been calculated. Intermediate values are loaded from storage as needed and the final values are written back.

A key hurdle here is to precisely identify the point in time when all parallel intermediate calculations are completed. This dependency introduces a notion of state to the system, as serverless computation has no concept of state an external service needs to be used.

The orchestration component defines the sequence of tasks; all executions are automatically triggered, each step is tracked and retried in the case of error. The corresponding orchestration plan, or state machine, for the distributed Strassen Algorithm, consists of a parallel task encompassing seven intermediate multiplication functions with a consecutive collection function. This pattern is hereafter referred to as the *Strassen Compute Unit* (SCU).

The same state machine can be used for different matrix sizes until the multiplication functions either hit their memory or time limit. The collector task can easily run as 2 or 4 distinct parallel instances to further speed up computation, or when a single collector instance tends to run out of disk space or main memory.

Determining which portions of the factors have to be loaded is a challenge, since the square input matrices of element count n need to be divided into up to four square sub-matrices of width $n_{sub} = n/2$. Each instance needs to find and load the required matrix partitions and possibly stitch together the pieces if partition size and size of objects in storage are not the same.

2) Workload Parallelization: The basic SCU is utilizing seven instances to perform matrix multiplication parallelization. However to archive **(I) Scalability**, further we need to distribute our workload for arbitrary matrix sizes. Even though the Strassen algorithm can be applied recursively, the improvement will not work well for a distributed system as [9] already suggested.

It is, therefore, necessary to partition the workload differently. Our solution keeps the above described SCU as a fixed-size state machine as the basic unit of multiplication and reapplies multiple times. Work will be distributed across the necessary number of SCUs which are coordinated through a surrounding parallel task – we call this a split (see figure 2).

There the only problem with this hierarchical approach is that the subproblems need to be independent and therefore the input matrices should not overlap and must always remain square. If at the edges, square multiplication is not possible, we apply zero padding.

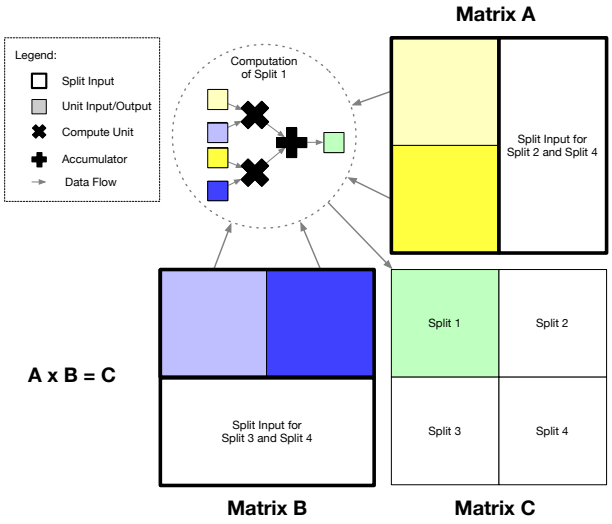

Figure 2. Visualization of a matrix multiple using our algorithm.

In summary, we designed **(I) Scalability** not by applying Strassen recursively but instead by dividing the workload into subproblems. This introduced two problems. First, we now have to aggregate all sub results in multiple stages. Second, we have to coordinate the execution of these steps.

B. SCU Coordination

The partitioning of our workload using multiple SCUs allows us to calculate larger matrices in a more performant way. However, this partitioning requires us to consider how we coordinate each SCU, how we setup inputs and how we accumulate all results to a final resulting matrix.

For this service orchestration, we are using a state machine engine offered by Amazon. This orchestration engine ensures that a collector function is only triggered after all subproblems are solved. Relying on a choreography instead of the orchestration function would have required a high amount of complexity which is why we instead utilized the state machine solution.

To initiate a matrix operation, we first invoke the setup function with the necessary parameters and matrix descriptors. An input matrix descriptor includes bucket, 'folder', matrix dimension and object size in storage. The setup function will use the matrix descriptors and performance parameters to create an appropriate state machine and start the required executions. The orchestration component will distribute the necessary meta-data to individual function instances, see also figure 3. After computation, the result can be found at the stated location and will be stored with objects of the defined block size.

Besides allowing to tune scalability, performance, and cost through the computation, the selected storage service and

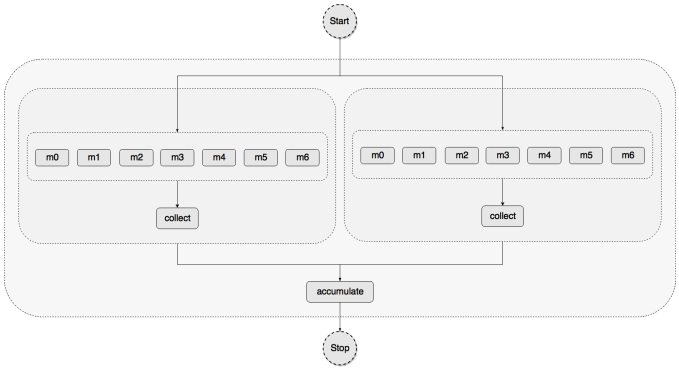

Figure 3. Visual workflow of calculation state machine with two SCUs.

schema, as well as the orchestration service, can have an impact. However, we only focus on the computation aspects of this design.

IV. System Implementation

The design described above uses AWS S3 for storage, AWS Lambda for compute and AWS Step Functions for orchestration. We implemented the AWS lambda function using python, which already offered good libraries for handling matrix data and performing local multiplication.

A. Orchestration with Step Functions

Step Functions[8] is a new service added to Amazon's Serverless Platform in December 2016[9]. Step Functions addresses obstacles that surfaced with AWS Lambda, such as the timing of invocations, waiting for periods of time and coordination of parallel executions, waiting for the completion of multiple running jobs, thus enabling more complex workloads besides simple event-driven pipelines.

Each step of an application is triggered, tracked and even retried when errors occur, assuring that applications execute as expected and in the correct order. To aid debugging and diagnostics, logs are kept for every step of the process. Figure 3 shows this process.

An added benefit is that step functions can be created and deployed in code through the definition of a state machine using Amazon's JSON-based States Language[10]. We used this language to build a tool that creates tailor-made execution plans for a given input matrix.

B. Matrix representation and storage

We utilize the block-schema for matrix storage. This data schema can best be supported by an object storage layer such as S3. Similar to the work by E. Jonas et al. [10] we discovered that S3 is surprisingly fast as a storage backend for AWS Lambda functions. We have to, however, consider the block size to achieve an optimal balance between I/O and compute of our implementation. As the functions are limited in local storage and memory, only small files can be read from the object store, therefore the network impacts remain also small.

Matrices are divided into blocks of a fixed element count. With the exception of edge blocks, all blocks are square and have equal block height and width β so that $\beta * \beta =$ block size.

If the dimensions of the matrix are not multiples of the block width, edge blocks are left non-square and thus of a smaller element count. Lambda instances will add zero padding as needed to perform matrix multiplication.

Though the size of blocks is virtually adjustable at will, consideration of final matrix size and degree of parallelization in computation, controlled through split size σ, will keep unnecessary overhead and corresponding cost at a minimum, and improve performance. While Lambda instances will stitch together blocks as necessary, it is recommended to match the block width β with the anticipated split width ω so that block size $= f(\omega) = \frac{\omega}{2^n}, n \in \mathbb{N}^+$ because *splits* are divided into 4 equally sized portions by SCUs. Smaller block sizes will result in an increased number of request and reduce loading throughput, while larger block sizes will add overhead as more elements than required are loaded. For maximum efficiency, the block size is $\frac{\sigma}{4}$ or block width $\beta = \frac{1}{2}\omega$ respectively.

To further increase the performance, limited caching of the data can be used. AWS lambda functions can utilize two limited means of caching, in-memory caching and on-disk caching. However, due to the random lifecycle of function containers, we can only employ an optimistic strategy by storing tempory downloaded files on disk and check if they are available before download.

V. Evaluation

We conducted three experiments to evaluate how well our system meets our design requirements (see section II) and compares to state-of-the-art distributed computing frameworks offered as a fully managed cloud service. Precisely, we evaluated scalability (E1) and tuneability (E2) of our system and compared performance and job costs to deployments of Apache Spark[11] and Hadoop MapReduce[12] clusters managed by the cloud service Amazon EMR[13] (E3). We designed and executed based on methodologies described in [4], [5].

A. Experiment Design

1) Deployments: For all experiments, we use the AWS region `eu-central-1` and S3 as storage backend inputs

[8] aws.amazon.com/step-functions

[9] https://aws.amazon.com/about-aws/whats-new/2016/12/introducing-aws-step-functions/

[10] https://states-language.net/spec.html

[11] http://spark.apache.org

[12] http://hadoop.apache.org

[13] http://aws.amazon.com/emr

and results. We distinguish between four deployments D0-3.

D0 refers to a deployment that serves as a baseline for comparison. It uses a specialized library for matrix multiplication[14] wrapped in a single Lambda function.

D1 refers to a deployment of our system. We configure multiplier functions with `1024MB` memory and collector functions and accumulator functions with `1536 MB` memory. The execution time of all functions is limited to `300 seconds`. The upper limit of functions that can execute in parallel is `1000`, i.e., the default limit for an AWS user.

D2 and **D3** refer to a deployment of a Hadoop and Spark cluster, respectively. D3 and D4 are backed by EC2 instances of type `m4.large`. Both use a single master node in each cluster.

2) Metrics: For all experiments, we measured performance and cost metrics. We measured the performance metrics *job runtime* and *function runtime* and the cost metric *job costs*.

- **Job Runtime** ($R_{job} \in \mathbb{N}$): Job runtime, i.e., job latency, is the difference of job end and job start in seconds. It describes the required overall time for a matrix multiplication.
- **Function Runtime** ($R_{fuc} \in \mathbb{N}$): Function runtime, i.e., function latency is the difference of the end and start of executing a function handler of a function in milliseconds.
- **Job Costs** ($C_{job} \in \mathbb{N}$): Job costs describes the marginal costs per matrix multiplication in terms of aggregated infrastructure costs billed by a cloud provider in USD.

For EMR-based deployments (D2-D3), job runtime does not include setup and teardown times for the cluster.

We use two different approaches to instantiate job costs for computations running on on Lambda- and EMR-based deployments.

For Lambda experiments, we base our cost modellling approaches on [6]. C_{job} is the sum of infrastructure costs for three services: Lambda costs C_λ, Step Function costs C_{sfn}, and S3 costs C_{s3}.

$$C_{job}(C_\lambda, C_{\text{sfn}}, C_{\text{s3}}) = C_\lambda + C_{\text{sfn}} + C_{\text{s3}} \tag{1}$$

Lambda costs C_λ is a function of the sum of funtion runtimes $r \in \mathbb{N}$, the number of function invocations $i \in \mathbb{N}$, and memory per invocation $m \in \mathbb{N}^+$ in MB:

$$C_\lambda(r, i, m) = 10^{-6}(\frac{0.01667rm}{1024} + \frac{i}{5}) \tag{2}$$

Step Function costs C_{sfn} is a function of the number of state transitions $t \in \mathbb{N}$:

$$C_{\text{sfn}}(t) = t * 0.000025 \tag{3}$$

Since data transfer in a region is free, only the number of PUT requests $p \in \mathbb{N}$ and GET requests $g \in \mathbb{N}$ are relevant for S3 cost calculation:

$$C_{\text{s3}}(p, g) = 10^{-5}(0.54p + 0.043g) \tag{4}$$

For our Spark and Hadoop experiments, C_{job} is the sum of infrastructure costs for the three used services: EC2, EMR, and S3. Total costs per job C_{server} is a function of cluster size $n \in \mathbb{N}^+$ [#nodes] and runtime $r \in \mathbb{N}$ [s]. We assume an EC2 instance cost of $C_{\text{ec2}} = 0.12$ USD/h and an EMR cost of $C_{\text{emr}} = 0.03$ USD/h.

$$C_{\text{server}}(r, n) = n * (r * C_{\text{ec2}}) + r * C_{\text{emr}} + C_{\text{s3}} \tag{5}$$

3) Parameters: We controlled paramters in all experiments and changed one or more of the following parameters. We describe each parameter with its corresponding value domain.

- **Matrix Size** ($M \in \{16, 64, 128\}$): Matrix size represents the number of elements in each of the input matrices in millions. Therefore, matrix size adjusts the difficulty or complexity of a multiplication. For example, $M = 16$ refers to the multiplication of two input matrices that each contain 16 million elements.
- **Split Size** ($P \in \{4, 16, 64\}$): Split size represents the number of elements in the result matrix that are computed by a single accumulator. Thus, a smaller split size indicates a higher parallalizm.
- **Units** ($U \in \{1, 2, 4\}$): Units represents the number of SCUs that are used to compute a single split. A larger number of units indicates a higher parallalizm.
- **Workers** ($W \in \{3, 15\}$): Workers represents the number of worker nodes in a Hadoop or Spark cluster.

B. Results

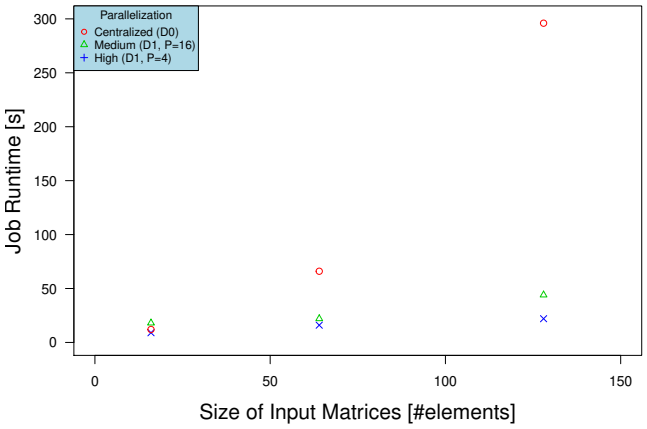

Figure 4. Scatter Plot Illustrating The Relationship Between Job Runtime R_{job} And Matrix Size M For Deployment D, Split Size P, And Units $U = 2$.

1) Scalability Experiments : The results for our scalability experiments (see figure 4) indicate that, for our serverless

[14]https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

approach, job runtime increases linearly with matrix size for a constant split size. For almost all multiplications, a smaller split size results in a faster job runtime. For the smallest matrix size, i.e., 16M elements, we measured runtimes of 12s for D0, 18s for $M = 16$, and 9s for $M = 4$. The results indicate that for very small matrix sizes, the centralized baseline approach can outperform our approach due to the additional overheads for distribution. However, for jobs of a matrix size greater than $128M$ elements, the centralized approach constantly fails due to the upper bound to function runtime of 300s. Table I illustrates the internal scaling behaviour of our serverless approach. Our results indicate that the number of functions increases linearly with the matrix size under constant split size. Thus, for M=256 and P=4, the number of multiplier functions becomes greater than 1000.

P\M	16M	64M	128M
16M	1/2/17	4/ 8/ 68	8/16/136
4M	4/8/68	16/32/272	32/64/544

Table I
TRIPEL (#SPLITS/#UNITS/#FUNCTIONS) FOR COMPUTING C=A*B WITH M DENOTING THE NUMBER OF ELEMENTS IN EACH A AND B AND P DENOTING THE NUMBER OF ELEMENTS IN RESULT MATRIX C COMPUTED BY A SINGLE SPLIT.

In our previous experiment, we used a constant number of units $U = 2$. Table II presents function runtimes of accumulator functions for different numbers of units that provide input to the accumulator and the number of elements per unit. Our measurements show that function runtime of an accumulator increases exponentially with the number of units. Our results indicate that function runtime of accumulators can quickly become greater than 300s for larger values of U.

U\N	1M	2M	4M	8M	16M	32M	64M
1	-	-	2.0	-	5.3	-	21.0
2	-	2.9	-	9.1	-	38.4	-
4	4.8	-	12.0	-	51.3	-	-

Table II
AVERAGE FUNCTION RUNTIME [S] OF ACCUMULATOR FUNCTIONS FOR DIFFERENT NUMBERS OF UNITS U AND OUTPUT ELEMENTS PER UNIT N.

2) Tunability : Figure 5 illustrates the results for our tunability experiment. For the same matrix size, our measurements for different split sizes rank all differently on a paretro-front of the trade-off between job runtime and job cost. Precisely, for all matrix sizes greater than $M = 16$, a smaller split size results in higher job costs and lower job runtime. Therefore, our results indicate that for sufficiently complex jobs split size is an effective and easy to predict configuration parameter that allows application developers specifying performance-cost preferences.

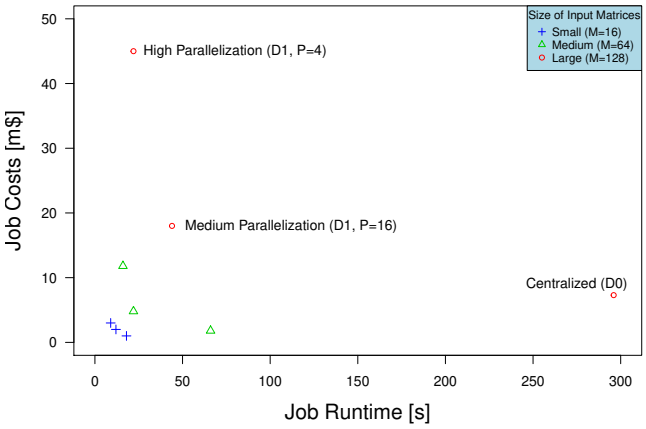

Figure 5. Scatter Plot Illustrating the Relationship Between Job Costs C_{job} and Job Runtime R_{job} For Matrix Size M, Deployment D, Split Size P, And Units $U = 2$.

D\M	**16M**	**64M**	**128M**
D1 (P=16M)	18s	22s	44s
D1 (P=4M)	9s	16s	22s
D2 (W=4)	133s	214s	-
D2 (W=16)	64s	135s	-
D3 (W=4)	23s	42s	62s
D3 (W=16)	10s	11s	16s

Table III
COMPARISON OF JOB RUNTIMES R_{job} [S] FOR DIFFERENT DEPLOYMENTS D AND MATRIX SIZES M.

3) Hadoop and Spark Comparison : Table III and table IV show our results for the comparison of deployments D1-D3. As can be seen in table III, the distributed parallel Lambda solution is comparable to a tuned Spark or Hadoop cluster. We measure significantly lower job runtimes for both deployments D1 and D3 compared to D2. For both split sizes $U = 16$ and $U = 4$, D1 achieves lower job runtimes than a Spark cluster with four worker nodes ($W = 4$). For split size $U = 4$, we measure similar job runtimes to a Spark cluster with 16 worker nodes ($W = 16$). We could produce reliable results for experiment **D2** with $M = 128$. Repeated experiments for this setup took over an hour even though we expected the results in significantly less time, however, not all runs of this experiment showed that behavior.

We measure significantly lower job costs for all deployments of D1 compared to deployments D2 and D3 (see table IV.

C. Discussion

Our experiments (see V-B1 - V-B2) show how the different system parameters affect scalability, performance, and job costs.

We compared two scaling strategies based on splits and

D\M	16M	64M	128M
D1 (P=16M)	$0.0038	$0.0048	$0.018
D1 (P=4M)	$0.0016	$0.0118	$0.045
D2 (W=4)	$0.63	$0.69	-
D2 (W=16)	$0.62	$0.69	-
D3 (W=4)	$0.56	$0.58	$0.59
D3 (W=16)	$0.56	$0.56	$0.57

Table IV
COMPARISON OF JOB COSTS C_{job} [USD] FOR DIFFERENT DEPLOYMENTS D AND MATRIX SIZES M.

units. Both strategies increase parallelism. However, a large number of splits put a higher load on a single accumulator function. At the same time, an accumulator function is bound to a 300s runtime limitation. Therefore, the accumulator function represents a potential bottleneck in the current version of our system for a split-based scaling strategy. Consequently, our results indicate that scaling approaches should first scale by increasing the number of splits instead of the number of units per split.

Our experiments indicate that the parameter split size is an effective tuning-knob to configure the trade-off between job runtime and job costs. In particular, small split sizes can significantly shorten job runtime. However, we observed the limitations to this speedup approach. Cloud infrastructure services typically expose a pre-configured soft limit for the number of function instances that can be executed in parallel, e.g., 1000 for AWS Lambda. A small split size for a large input matrix can result in a high number of function instances (see table I) that are executed in parallel. As a result, jobs can fail and job runtime and job costs can increase. Therefore, setting appropriate configurations for function limits and split size are two remaining management tasks in the current version of our system. For future work, we plan to automatically configure AWS account limits, function memory size, and split size based on application level metrics provided by a user.

Our experiment results indicate that serverless big data processing can compete, match, and outperform cluster-based state-of-the-art distributed compute frameworks in terms of performance, scale and costs (see V-B3). At the same time, we argue that decision-makers should be aware of a number of open challenges in selecting the best infrastructure platform for their processing needs. First, we argue that a direct fair performance and cost comparison of cluster-based and serverless systems remains a difficult task. For example, the serverless approach has significantly lower startup or teardown times, e.g., between 1 and 5 minutes per run for **D2** and **D3**. Therefore, it provides potential benefits in application scenarios with ad-hoc jobs. On the other hand, cost-efficiency of cluster-based systems could potentially increase under mixed workloads. Second, we argue that additional methods, tools, and experiments are needed to support decision makers in selecting the best infrastructure platform for processing needs at hand.

Finally, we argue that our proof of concept matches the requirements defined in section II. Thus, for specific tasks, serverless infrastructure should be considered a new relevant tool in the big data processing toolbox.

VI. RELATED WORK

Utilizing a serverless infrastructure for big data applications is still new, and thus, there are many open questions and issues. Early adopters have already started to investigate the serverless paradigm [11]–[13] for big data processing.

There are not many publications that look at the concrete benefits of using a serverless infrastructure solely as the data processing platform. Flit by Y. Kim and J. Lin [14] is the most closely related work to our as it also utilizes the serverless infrastructure and compares it with Apache Spark. Their main contribution is an extension of work done by E. Jonas et al. in [15] with the PySpark runtime, enabling the use of a serverless infrastructure as if it were a spark compute cluster. Even though their approach already offers practical ways to use a serverless infrastructure for big data applications, it still only shows parts of the benefits and insights needed to make knowledgeable decisions. Furthermore, their approach of adopting a programming model designed for cluster operations creates similarly complex configuration problems as a non-serverless approach.

There is a plethora of work on distributed big data frameworks among them enhancement for Spark [16] and Hadoop [17]. Work on cloud computing integration for these frameworks has also been researched in many ways. However, the serverless infrastructure creates new opportunities [18] and abilities that neither of the available frameworks has yet adapted.

Our work uses the matrix multiplication problem to showcase how a serverless application can be built. There is related previous work that used this distributed computation problem also. Particularly, the work done by R.A. Van De Geijn et al. [19] and HAMA [20] done by S. Son et al. showed potential ways to distribute this computation problem.

VII. CONCLUSION

In this paper, we explored the use of serverless infrastructure for big data processing using matrix multiplication as example. We formulated generic requirements for designing serverless data processing applications and performed a series of experiments.

We showed that the utilization of serverless infrastructure indeed can lower operational and infrastructure costs without compromising established system qualities. Our experimental results indicate that serverless implementations can situationally compete, match, and even outperform cluster-based distributed compute frameworks regarding performance and scalability.

Our prototype provides a single tuning-knob that allows application developers to configure the cost/performance trade-off according to requirements at hand. Thus, serverless solution can significantly reduce the entry barrier for new developers both in terms of costs and lowered complexity of configuring a sophisticated big data solution stack.

For future work, we plan on fully automating the configuration of our prototype by introducing a preprocessing step to analyze the input data. Furthermore, more experiments on serverless implementations will help to further validate our design guidelines.

ACKNOWLEDGMENTS

The work in this paper was performed in the context of the DITAS and BloGPV.Blossom project. DITAS is partially supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 731945. BloGPV.Blossom is partially funded by the Germany Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. 01MD18001E. The authors assume responsibility for the content.

REFERENCES

[1] V. Markl, "Mosaics in Big Data: Stratosphere, Apache Flink, and Beyond." ACM Press, 2018, pp. 7–13, 00000.

[2] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of Function-as-a-Service software development in industrial practice."

[3] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, no. Figure 1, pp. 14–19, 2016.

[4] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[5] M. Klems, "Experiment-driven evaluation of cloud-based distributed systems," dissertation, 2016.

[6] J. Kuhlenkamp and M. Klems, "Costradamus: A Cost-Tracing System for Cloud-based Software Services," in *Proceedings of the 15th International Conference on Service Oriented Computing (ICSOC17)*. Springer, 2017.

[7] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1219–1230, 2014.

[8] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai, "BenchFoundry: A Benchmarking Framework for Cloud Storage Services," in *Proceedings of the 15th International Conference on Service Oriented Computing (ICSOC17)*. Springer, 2017.

[9] N. J. Higham, "Exploiting fast matrix multiplication within the level 3 blas," *ACM Transactions on Mathematical Software.*, vol. 16, no. 4, pp. 352–368, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98267.98290

[10] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," pp. 1–22, 2017. [Online]. Available: http://dx.doi.org/10.13140/RG.2.2.15007.87206

[11] B. Congdon, "Introducing Corral: A Serverless MapReduce Framework," May 2018. [Online]. Available: http://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/

[12] Sunil Mallya, "Ad Hoc Big Data Processing Made Simple with Serverless MapReduce," Nov. 2016. [Online]. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[13] Bhaskar Das, "Can Serverless computing reshape big data and data science?" 2018. [Online]. Available: https://dashbird.io/blog/serverless-computing-reshape-big-data-data-science/

[14] Y. Kim and J. Lin, "Serverless Data Analytics with Flint," *arXiv:1803.06354 [cs]*, Mar. 2018.

[15] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%." ACM Press, 2017, pp. 445–451, 00041.

[16] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling Spark in the Real World: Performance and Usability," p. 4, 2016, 00053.

[17] Y. Guo, J. Rao, D. Cheng, and X. Zhou, "iShuffle: Improving Hadoop Performance with Shuffle-on-Write," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1649–1662, Jun. 2017, 00070.

[18] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," pp. 1–20, 2017. [Online]. Available: http://arxiv.org/abs/1706.03178

[19] R. A. van de Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," Austin, TX, USA, Tech. Rep., 1995.

[20] S. Seo, E. J. Yoon, J. Kim, S. Jin, J. S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the MapReduce framework," *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, pp. 721–726, 2010.