MapReduce na AWS: Uma Análise de Custos Computacionais Utilizando os Serviços FaaS e IaaS*

Ronald Campbell[1], **Alan L. Nunes**[1], **Cristina Boeres**[1], **Lúcia Maria de Assumpção Drummond**[1]

[1] Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ – Brasil

{ronald,alan_lira}@id.uff.br, {boeres,lucia}@ic.uff.br

Resumo. *Serviços da nuvem oferecem facilidades computacionais bem estabelecidas. Na busca de eficiência para executar aplicações do tipo MapReduce, que lidam com grandes volumes de dados, baixos custos monetários também são almejados. Para delinear os benefícios de diferentes serviços cloud, realizamos uma análise exploratória dos tempos e custos para a execução de uma aplicação MapReduce na nuvem pública da Amazon, a AWS. A partir de implementações com os frameworks Spark e MARLA sob os serviços EC2 e Lambda, respectivamente, apresentamos os impactos associados às quantidades e tipos de recursos escolhidos. Os resultados sugeriram o ambiente MARLA Lambda como o mais rápido e o Spark EC2 como o mais econômico.*

1. Introdução

A popularização do ambiente de nuvens computacionais é uma realidade, principalmente no que tange ao compartilhamento de dados e aos serviços de uso geral. Investimentos têm sido realizados, tanto por provedores quanto por clientes [Muniswamaiah et al. 2019], para torná-lo uma solução promissora para a execução de aplicações que processam grandes volumes de dados — *Big Data*. Este paradigma computacional possui diversas vantagens quando comparado às infraestruturas dedicadas, principalmente pela significativa redução de custos operacionais relacionados a gastos energéticos, licenças de software, e obsolescência de hardware. Assim, os clientes priorizam o desenvolvimento e a implantação de aplicações sem a preocupação de gerenciamento e manutenção da infraestrutura computacional. Os provedores de nuvens fornecem infraestrutura, plataformas e software através de acordos de nível de serviço (*SLA*s) com os consumidores.

Por outro lado, diversos desafios devem ser superados para preencher a lacuna entre os desempenhos obtidos em uma infraestrutura dedicada e na nuvem. Por exemplo, sobrecargas em relação ao uso de CPU, memória, rede, e disco introduzidos pela camada de virtualização e arquitetura *multi-tenant* (*i.e.* compartilhamento de recursos de um servidor físico por parte de diferentes clientes) afetam o desempenho de execução de aplicações *Big Data* na nuvem, em comparação com uma infraestrutura dedicada. Embora a camada de virtualização forneça um nível razoável de proteção e isolamento, alguns recursos (*e.g.*, *cache* e memória principal) são compartilhados entre as aplicações

*Esta pesquisa é amparada pelo projeto CNPq/AWS 440014/2020-4, pelo *Programa Institucional de Internacionalização* (PrInt) da CAPES (processo de número 88887.310261/2018-00), pelo Projeto Universal/CNPq n^o 404087/2021-3 e pelo Projeto CNE/FAPERJ n^o E-26/201.012/2022(271103).

que estão sendo executadas em máquinas virtuais instanciadas em um mesmo servidor físico [Awaysheh et al. 2021, Kapil et al. 2022].

Desde 2010[1] tem sido disponibilizado o serviço de nuvem "sem servidor" ou *Function as a Service* (*FaaS*), em que os clientes não se preocupam ativamente com planejamento, configuração, gerenciamento, e dimensionamento de recursos. Neste modelo de serviço, a computação é realizada em *bursts*, e seus resultados são persistidos no armazenamento definido pelo usuário. Quando uma aplicação não está sendo executada, nenhum recurso é alocado dinamicamente pelo provedor. Ao contrário do modelo *Infrastructure as a Service* (*IaaS*), em que o usuário paga pelas instâncias virtuais por segundo (*e.g.*, máquinas virtuais), independentemente se estiverem ou não em utilização enquanto instanciadas, o custo do modelo *FaaS* é de fato *pay-per-use*, pois o usuário pagará apenas pelos recursos realmente consumidos durante a execução das funções implementadas para suas aplicações. Através de uma analogia[2] onde o usuário deseja percorrer um determinado trajeto, o modelo *IaaS* pode ser comparado ao aluguel de um carro (cujas manutenções exigidas durante o período de locação do carro são de responsabilidade do locatário), e o modelo *FaaS* pode ser comparado a utilização do *Uber*. Dentre os provedores pioneiros e populares de computação *serverless*, com alto grau de paralelismo a partir da invocação simultânea de até milhares de funções definidas em linguagens de programação suportadas pelo serviço, podemos citar *Amazon Lambda*, *Google Cloud Functions*, e *Microsoft Azure Functions*.

O desenvolvimento do modelo *MapReduce*, cujo o processamento é concentrado nas fases essenciais *Map* e *Reduce*, favoreceu a implementação de diversas ferramentas escaláveis e confiáveis de processamento para grandes volumes de dados. Dentre os *frameworks open-source* disponíveis, podemos citar *Apache Hadoop* [Shvachko et al. 2010] e *Apache Spark* [Zaharia et al. 2012]. Ambos utilizam um *cluster* de nós computacionais para o gerenciamento e execução de aplicações *MapReduce*. Desta forma, o modelo *IaaS* oferecido por provedores de nuvens públicas pode ser utilizado para a construção de tais *clusters* de instâncias virtuais, encontrados em, por exemplo, *Amazon Elastic Compute Cloud* (*EC2*)[3], *Microsoft Azure Virtual Machines*[4], e *Google Compute Engine* (*GCE*)[5]. Além do clássico *Hadoop Distributed File System* (*HDFS*), *Hadoop* e *Spark* suportam diversos modos de armazenamento, *e.g.*, *Amazon Simple Storage Service* (*S3*)[6], que é um serviço escalável, seguro e de alta disponibilidade para o armazenamento de objetos.

Mais recentemente, foi proposto *MARLA* (acrônimo para *MApReduce on LAmbda*) [Giménez-Alventosa et al. 2019], ferramenta que permite a configuração do modelo *serverless* para a execução de *MapReduce* no *Amazon Lambda*. O usuário implementa a lógica computacional das funções *Lambda* para cada fase do modelo *MapReduce* através dos agentes *Mappers* e *Reducers*, que são disparados a partir do carregamento dos dados de entrada. Atualmente, o *MARLA* utiliza exclusivamente o sistema de armazenamento *Amazon S3* para o armazenamento dos dados de entrada e saída de cada aplicação.

O objetivo deste trabalho é avaliar a execução da aplicação clássica de *MapReduce*

[1] https://techcrunch.com/2010/07/19/picloud-launches-serverless-computing-platform-to-the-public/
[2] https://medium.com/nerd-for-tech/1-cloud-series-the-iaas-paas-saas-8faa46124722
[3] https://aws.amazon.com/pt/ec2/
[4] https://azure.microsoft.com/pt-br/services/virtual-machines/
[5] https://cloud.google.com/compute
[6] https://aws.amazon.com/pt/s3/

Word Count, que lê um arquivo de texto e contabiliza a frequência de cada palavra contida nele, considerando os serviços *FaaS* e *IaaS* da *AWS*. Serão utilizados os frameworks *Spark* sobre um *cluster* de máquinas virtuais otimizadas para memória no *Amazon EC2*, e o *MARLA*, sobre um conjunto de funções definidas no *Amazon Lambda*. Para ambos os ambientes, o *Amazon S3* será especificado como origem e destino dos dados. Para fins de avaliação, serão analisados os tempos de execução e custos financeiros associados a diversos cenários para a aplicação *Word Count*, nos quais serão variados os volumes dos dados de entrada, a quantidade de *Mappers* e *Reducers*, bem como a quantidade de memória alocada por função no *Amazon Lambda*, de modo a se delinear as vantagens e limitações deste modelo de serviço na resolução desta categoria de problema, quando comparado ao ambiente *Spark* no *Amazon EC2*.

O restante deste artigo está organizado da seguinte forma. A Seção 2 expõe os conceitos básicos de nuvens computacionais, da *AWS*, e do modelo *MapReduce*. A Seção 3 destaca os trabalhos relacionados. A Seção 4 descreve a implementação da aplicação *Word Count* na *AWS*, em versões para os *frameworks MARLA* e *Spark*. A Seção 5 apresenta a avaliação experimental e, por fim, as conclusões são expostas na Seção 6.

2. Conceitos Preliminares

Este trabalho concentra esforços em identificar vantagens de executar aplicações *MapReduce* nos modelos *FaaS* e *IaaS* em nuvens. Assim, esta seção apresenta os conceitos básicos de nuvens computacionais, da nuvem pública da *Amazon* (*AWS*), e do modelo *MapReduce*, um dos mais utilizados para *Big Data*.

2.1. Nuvens Computacionais

Em nuvens computacionais, uma gama de recursos físicos e virtuais pode ser atribuída dinamicamente de acordo com a demanda dos clientes, parecendo muitas vezes ilimitados em termos de tempo ou quantidade. Dos modos de implantação, as nuvens públicas são as mais populares, sendo *Amazon Web Services* (33%), *Microsoft Azure* (21%) e *Google Cloud* (8%) os três provedores de serviço mais utilizados em 2022[7]. *Amazon Web Services* (*AWS*) é, atualmente, a maior plataforma de computação em nuvem, abrangendo 84 zonas de disponibilidade em 26 regiões geográficas no mundo[8], e oferece 227 categorias de serviços[9], tais como: *Amazon Elastic Compute Cloud* (*EC2*), *Amazon Lambda* e *Amazon Simple Storage Service* (*S3*).

Amazon Elastic Compute Cloud (*EC2*) é um serviço *IaaS* que oferece mais de 500 opções de máquinas virtuais para atender diversos *workloads*, sendo suas instâncias de máquinas virtuais (MVs) agrupadas nos tipos *General Purpose*, *Compute Optimized*, *Memory Optimized*, *Storage Optimized*, e *Accelerated Computing*. Também são oferecidas instâncias *bare-metal* que fornecem acesso direto ao processador e à memória do servidor para a execução de aplicações em ambientes não virtualizados ou para aplicações que implementam o próprio *Hypervisor*. Além da capacidade de computação segura e redimensionável que dispensa o investimento em *hardware* físico próprio, o *EC2* permite o controle dos recursos em utilização. O faturamento de instâncias ocorre em incrementos

[7] https://www.channele2e.com/news/cloud-market-share-amazon-aws-microsoft-azure-google/
[8] https://aws.amazon.com/pt/about-aws/global-infrastructure/
[9] https://aws.amazon.com/pt/products/

de segundo, hora ou mês de uso, dependendo do tipo da instância, região de localização e mercado de preços, cujas principais opções são *On-Demand*, *Reserved* e *Spot*. MVs *on-demand* costumam ser mais caras, mas oferecem estabilidade de uso, enquanto que *Spots* têm descontos atraentes, porém sujeitas a interrupções a qualquer momento com durações indeterminadas.

Amazon Lambda é um serviço *FaaS* (*Function as a Service*) que executa código em resposta a eventos utilizando uma infraestrutura de computação de alta disponibilidade de forma transparente seguindo o modelo "*serverless*" (*i.e.*, os servidores são utilizados, mas sem a necessidade de seu gerenciamento por parte do usuário). Eventos podem incluir mudanças de estado ou atualizações (como por exemplo, adição de item à um carrinho de compras em um *website* de comércio eletrônico). O *Lambda* invoca o código do cliente a partir da ocorrência de um evento suportado pelo AWS Lambda, escalando-o automaticamente para comportar a taxa de solicitações, que pode ser ilimitada. O cliente paga, em incrementos de um milissegundo, apenas pelas solicitações atendidas e pelo tempo de computação necessário para executar o código e não pela unidade do servidor que o executou. Atualmente, o tempo máximo de execução de uma função Lambda é de 15 minutos. Após este prazo, a execução é encerrada. De acordo com a quantidade de memória definida pelo cliente para a execução de suas funções no *Lambda*, são alocadas capacidades proporcionais de *CPU*, largura de banda de rede, e entrada/saída de disco.

Amazon Simple Storage Service (*S3*) é um serviço *STaaS* (*STorage as a Service*) que oferece escalabilidade, disponibilidade de dados, segurança, e desempenho para o armazenamento de objetos na *Cloud*. Os dados são armazenados como objetos em recursos chamados *buckets* e um único objeto pode ocupar até 5 *terabytes* de volume. Os objetos podem ser acessados por meio de *S3 Access Points* ou diretamente pelo *hostname* do *bucket*. O *Amazon S3* oferece funções para mover e armazenar dados em classes de armazenamento, configurar e impor controles de acesso aos dados, proteção contra usuários não autorizados, executar análise de *Big Data*, entre outros.

2.2. Modelo MapReduce

MapReduce [Dean and Ghemawat 2004] é um modelo de programação e uma implementação associada para o processamento de grandes conjuntos de dados. De um modo transparente ao usuário, *frameworks* atuais que implementam tal modelo lidam com paralelização de computação, distribuição e localidade de dados e balanceamento de carga, e lentidão ou falha de nós computacionais, permitindo assim o seu dimensionamento para *clusters* que compreendem até milhares de máquinas. O uso de um modelo funcional com operações de mapeamento e redução, especificadas pelo usuário, permite paralelizar vários cálculos computacionais. Os *frameworks* de *MapReduce* incorporam a tolerância a falhas a partir da reexecução de tarefas de nós computacionais (*Workers*) que falharam, por efeito da replicação de dados. O modelo *MapReduce* adota o seguinte fluxo de execução:

1. Os arquivos de entrada são divididos em M pedaços (*partitions*, *splits* ou *chunks*), sendo M definido pelo usuário;
2. São inicializadas um número de cópias da aplicação, sendo uma das cópias especificada como Mestre (*Master*), responsável por enviar tarefas (*Tasks*) a serem processadas pelas demais cópias, especificadas como Trabalhadores (*Workers*). Existem M

map tasks e *R reduce tasks* a serem atribuídas para os Trabalhadores, sendo *R* definido pelo usuário. O Mestre seleciona Trabalhadores ociosos e atribui, para cada um deles, tarefas *Map* ou *Reduce*;

3. Um Trabalhador que recebe uma tarefa *Map* (*i.e.*, *Mapper*) lê o conteúdo do seu pedaço do arquivo de entrada, analisa e gera pares do tipo *chave-valor*, enviando-os para a função *Map*, definida pelo usuário. Os pares *chave-valor* intermediários produzidos pela função *Map* são armazenados em *buffer* na memória;
4. Periodicamente, os pares intermediários são gravados no disco local, particionado em *R* regiões através de uma função de particionamento, (*e.g.*, *hash*(chave) *mod R*), definida pelo usuário. As localizações desses pares são enviadas para o Mestre, que fica responsável por encaminhá-las aos Trabalhadores designados para tarefas *Reduce* (*i.e.*, *Reducers*);
5. Quando um *Reducer* é notificado sobre essas localizações, utiliza chamadas de procedimento remoto para ler os dados armazenados em *buffer* nos discos locais dos *Mappers*, e após a leitura de todos esses dados, classifica as chaves intermediárias para agrupar todas as ocorrências por chave. Esta etapa de classificação é necessária, pois normalmente muitas chaves diferentes são mapeadas para a mesma tarefa *Reduce*. Se a quantidade de dados intermediários for maior que a memória disponível, então uma classificação externa será utilizada;
6. O *Reducer* itera sobre os dados intermediários classificados e, para cada chave única encontrada, passa a chave intermediária com o seu conjunto correspondente de valores para a função *Reduce* definida pelo usuário. A saída da função *Reduce* é anexada a um arquivo de saída final para cada partição de redução, totalizando *R* arquivos de saída no final da execução.

3. Trabalhos Relacionados

Em [Nunes et al. 2021] tratamos do provisionamento de recursos em nuvens para executar aplicações tipo *MapReduce* com *Spark*, visando a redução de custos. A aplicação *Diff Sequences Spark* foi implementada para comparar as sequências biológicas de entrada e gerar como saída as posições onde os nucleotídeos diferem entre si. Experimentos realizados na *AWS EC2* com até 540 sequências de *SARS-CoV-2* (*COVID-19*) demonstraram o custo-benefício de instâncias otimizadas para memória, mesmo em cenários de revogação de MVs do mercado *spot*.

[Kim and Lin 2018] apresentaram o *Flint*, um protótipo de mecanismo de execução do *Spark*, que fornece um modelo de custo *pay-per-use* através do *AWS Lambda*. *Flint* suporta a execução transparente de código escrito para *PySpark* (*i.e.*, a interface para *Spark* implementada em *Python*), de modo que o usuário não necessita construir um *cluster Spark* com MVs. A manipulação de dados intermediários no *Flint* é realizada por meio de um serviço de filas de mensagens, o *Amazon Simple Queue Service* (*SQS*). O desempenho do *Flint* foi comparado ao de um *cluster Spark* com MVs da *AWS EC2* (construído através da *Databricks' Unified Analytics Platform*), sendo este utilizado para a execução de códigos em *Python* (*PySpark*) e em *Scala* (*Native Spark*). Os resultados indicaram que *Flint* é particularmente preferível para o processamento de análises *ad hoc* e *exploratória de dados*. Note que, a execução de uma aplicação através deste framework incorpora a necessidade de implantação do ambiente *Spark*.

Em [Giménez-Alventosa et al. 2019], além do desenvolvimento do *MARLA*, foram apresentadas análises de desempenho entre *MARLA* e *Serverless Reference Architecture for MapReduce*[10] (*SRAM*) da *AWS*, visando uma avaliação dos benefícios e limitações do *AWS Lambda* como uma plataforma de computação geral e de execução de *jobs* paralelos dependentes (*e.g.*, aplicações*MapReduce*). Os resultados indicaram que o *AWS Lambda* apresenta um comportamento heterogêneo de desempenho, em termos de *CPU* e de rede, que impacta no tempo de execução de jobs *MapReduce*. Foram indicados alguns padrões de uso que podem ser adotados para mitigar a ocorrência de desempenho heterogêneo, tais como: i) O usuário deve garantir que a invocação da função *Lambda* será executada, mesmo quando uma MV com menor poder computacional for atribuída a ela; ii) O usuário deve supor que, eventualmente, a invocação de uma função *Lambda* causará um *timeout*, provocado por uma transferência de dados em uma rede mais lenta, o que demandará um correto tratamento desse tipo de erro no fluxo de execução da aplicação. Não foram apresentadas comparações diretas com *frameworks* populares de *MapReduce* que utilizem a infraestrutura da *AWS*.

Do nosso conhecimento, poucos trabalhos contemplam o uso do *MARLA*. A contribuição principal do nosso trabalho reside na comparação direta entre *MARLA* e *Spark* para a execução de uma aplicação *MapReduce* na *AWS*, visando a análise de tempos e custos monetários associados ao uso dos serviços *FaaS* e *IaaS*, respectivamente, delineando as vantagens e desvantagens de uso em cada cenário.

4. Aplicação Word Count

Word Count, também conhecida como *Word Frequency*, é uma aplicação clássica no domínio *MapReduce* que contabiliza o número de ocorrências de cada palavra única, dado um conjunto de arquivos de entrada especificado pelo usuário.

4.1. Implementação com MARLA

Após o preenchimento do arquivo de configurações do *MARLA* com informações básicas de autenticação na *AWS* e características das funções Lambda a serem criadas para a execução de tarefas, as etapas *Map* e *Reduce* são codificadas em *Python* pelo usuário:

1. ***Map***: Transformação de um bloco de texto em pares *chave-valor*, cujas chaves são palavras, e valores são números inteiros que representarão suas respectivas frequências. Nesta etapa, também é realizado um tratamento no arquivo de entrada, sendo removidas todas as ocorrências de vírgula no texto (que é um caractere especial para o processamento de dados parciais no *MARLA*). Os dados parciais gerados nesta fase são armazenados no *Amazon S3* e posteriormente recuperados para o processamento na fase *Reduce*;
2. ***Reduce***: Soma dos valores (frequências) contidos em todos os pares parciais resultantes da fase *Map* que possuam a mesma chave (palavra). Pares *chave-valor* agrupados por *chave* (palavra) serão gerados e, em seguida, armazenados no *Amazon S3*.

4.2. Implementação com Spark

O modelo *MapReduce* pode ser expresso no *framework Spark* através das seguintes operações de transformação: i) *flatMap* para a fase *Map*; e ii) *groupByKey* ou *reduceByKey* para a fase *Reduce* [Zaharia et al. 2012]. Como todas as transformações no *Spark*

[10] https://aws.amazon.com/pt/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

seguem a técnica de programação funcional *Lazy Evaluation*, que adia a computação até que um determinado ponto de execução exija os seus resultados, é essencial utilizar pelo menos uma operação de ação (*e.g.*, *collect*, *saveAsTextFile*) para ativar a computação dos dados. Em relação às estruturas de dados oferecidas pelo *Spark*, que são coleções imutáveis de objetos de dados distribuídos logicamente entre os nós computacionais (*Workers*) para processamento paralelo, foi adotado o *RDD* (*Resilient Distributed Dataset*). A execução do *Word Count* no *Spark* consiste das seguintes etapas:

1. Leitura do arquivo de entrada através da função *textFile*, gerando o RDD_1, cujos elementos são blocos de texto;
2. Definição do número de partições (*map tasks*) para o RDD_1 através das funções *coalesce* ou *repartition*, gerando o RDD_2;
3. Transformação do RDD_2 através da função *flatMap*, que fragmentará cada bloco de texto (elemento) em palavras, gerando o RDD_3;
4. Transformação do RDD_3 através da função *Map*, que mapeará cada elemento (palavra) em um par *chave-valor*, gerando o RDD_4;
5. Definição do número de partições (*reduce tasks*) para o RDD_4 através das funções *coalesce* ou *repartition*, gerando o RDD_5;
6. Transformação do RDD_5 através da função *reduceByKey*, que agrupará os pares *chave-valor* por chave (palavra) e somará todos os respectivos valores (frequências), gerando o RDD_6;
7. Gravação do RDD_6 em arquivos de texto, um arquivo por partição de redução, através da ação *saveAsTextFile*.

5. Resultados Experimentais na AWS

Os serviços *AWS Lambda* e *AWS EC2* foram utilizados para a execução das implementações com *MARLA* e com *Spark*, respectivamente. O *Amazon S3* foi utilizado para o armazenamento dos dados de entrada, intermediários (apenas para a versão com *MARLA*) e de saída. Foram coletados os tempos (em segundos) e custos monetários (em *USD*) de execução em diversos cenários para cada implementação. No caso da implementação com o *MARLA*, foi utilizado o serviço *CloudWatch*[11] que retorna as estimativas de interesse. Já na versão *Spark*, a implementação foi instrumentada para que tais estimativas fossem coletadas, visto que haveria dificuldade por parte do *CloudWatch* de resgatá-las, considerando as tarefas executadas internamente a um cluster *Spark*.

5.1. Análise dos Resultados com MARLA

Para a análise do *Word-Count-MARLA*, foram especificadas capacidades de memória, denotadas como **# Mem**, de 1024 *MB*, 2048 *MB*, 4096 *MB*, e 8192 *MB* por função Lambda. As quantidades de *Mappers* e de *Reducers*, denotadas por **# Map** e **# Red** respectivamente, foram variadas considerando as seguintes características: i) **# Map** variou em potências de 2 até 64, tendo em vista a capacidade mínima necessária para a execução da etapa *Map* no *MARLA* (que atualmente demanda um alto consumo de memória) sem que ocorresse um estouro de memória; e ii) **# Red** variou de 4 até 32, em potências de 2, sem

Tabela 1. Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (I): 2 GB Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
1024	**52**	**4**	1,13	51,25	87,38	139,76	0,05
		8	1,26	54,99	80,38	136,63	0,06
		16	1,62	54,81	42,62	99,05	0,06
		32	1,17	55,61	34,58	91,36	0,07
	64	**4**	1,15	43,42	90,89	135,46	0,05
		8	1,19	43,60	79,98	124,77	0,06
		16	1,12	41,57	43,89	86,58	0,06
		32	1,15	45,29	35,36	81,80	0,07
2048	**26**	**4**	1,19	61,22	40,52	102,93	0,06
		8	1,29	66,40	37,18	104,87	0,07
		6	1,20	61,35	23,45	86,00	0,07
	32	**4**	1,17	48,56	43,96	93,69	0,06
		8	1,22	49,47	40,86	91,55	0,06
		16	1,26	50,87	24,53	76,66	0,07
		32	1,19	50,52	19,80	71,51	0,07
	64	**4**	1,18	24,77	51,37	77,32	0,06
		8	1,24	24,93	46,70	72,87	0,07
		16	1,20	25,96	27,79	54,95	0,07
		32	1,11	26,30	21,66	49,07	0,08
4096	**12**	**4**	1,12	134,70	26,02	161,84	0,11
		8	1,21	135,63	24,07	160,91	0,12
	16	**4**	1,18	98,80	32,61	132,59	0,11
		8	1,19	100,45	30,63	132,27	0,12
		16	1,25	104,73	17,53	123,51	0,13
	32	**4**	1,15	49,13	44,39	94,67	0,12
		8	1,19	51,48	39,77	92,44	0,13
		16	1,26	51,29	27,48	80,03	0,14
		32	1,16	50,94	20,67	72,77	0,15
	64	**4**	1,17	24,71	50,08	75,96	0,12
		8	1,24	25,45	46,04	72,72	0,13
		16	1,20	28,09	29,16	58,45	0,15
		32	1,12	26,07	23,27	50,46	0,16
8192	**6**	**4**	1,14	281,38	12,19	294,71	0,23
	8	**4**	1,16	205,00	15,74	221,90	0,23
		8	1,23	217,85	13,88	232,95	0,25
	16	**4**	1,16	98,72	34,28	134,16	0,23
		8	1,19	100,49	30,89	132,57	0,25
		16	1,16	101,77	17,91	120,84	0,25
	32	**4**	1,15	49,38	48,18	98,71	0,24
		8	1,17	49,78	42,21	93,16	0,26
		16	1,14	50,24	25,20	76,58	0,27
		32	1,18	50,69	19,74	71,61	0,30
	64	**4**	1,22	24,56	53,64	79,42	0,24
		8	1,16	25,42	49,40	75,98	0,27
		16	1,18	25,66	29,83	56,67	0,28
		32	1,18	25,94	23,75	50,87	0,32

que excedesse o respectivo **# Map**. Para cada cenário formado, os volumes de dados de entrada (I) considerados foram 2 *GB* e 4 *GB*.

A **Tabela 1** apresenta as médias dos tempos de execução e respectivos custos monetários para a entrada de 2 *GB*, considerando cinco execuções por cenário. Foram também coletados os tempos associados aos agentes de cada fase no *MARLA*, *i.e.*, os tempos do *Coordinator* (responsável por realizar a divisão e distribuição dos dados de entrada), e os tempos das fases *Map* e *Reduce*, separadamente.

As funções Lambda com **# Mem** = 1024 *MB* apresentaram tempos significativamente maiores, em torno de 73%, quando comparadas aos cenários com **# Mem** = {2048, 4096, 8192} *MB*. Isto deve-se ao fato de que o Lambda aloca poder computacional (quantidade de vCPUs) proporcional à quantidade de memória provisionada [Malawski et al. 2020]. As funções com **# Mem** = {2048, 4096, 8192} alcançaram tempos similares entre si, por exemplo, para **# Map** = 64 e **# Mem** = {2048, 4096, 8192}. Note que em 2020 houve um aumento das capacidades computacionais por função Lambda.[12]

Vale ressaltar que **# Map** influencia o tempo de execução da fase *Reduce* para qualquer **# Mem**. Quanto mais *Mappers*, mais dados parciais são gerados, sendo estes posteriormente recuperados e processados pelos *Reducers*. Por outro lado, quanto menos *Mappers*, maior o risco de estouro de memória na etapa *Map* do *MARLA*. Os melhores tempos de execução foram obtidos com **# Map** = 64, e **# Red** = 32, para todos os cenários considerados. Por último, os comportamentos de execução inferidos para 2 *GB* de entrada foram observados de forma análoga para 4 *GB* de entrada.

5.2. Análise dos Resultados com Spark

Para a análise do *Word-Count-Spark*, foi selecionada a instância *r5.large* (com 2 *vCPUs* e 16 *GiB* de Memória) para a formação de diferentes tamanhos de cluster *Spark*, pois, segundo a *Amazon*[13], é um tipo de MV ideal para *caches* distribuídos na memória e análises de *Big Data*. Ademais, em [Nunes et al. 2021] foram observadas vantagens na utilização de MVs otimizadas para memória para este tipo de aplicação. O mercado *Spot* foi utilizado em todos os experimentos, de modo que o custo monetário para cada instância foi fixado em 0,048 *USD* por hora de uso. Foram desconsiderados os custos associados ao tempo de construção de cada cluster na *AWS EC2* e ao armazenamento e recuperação de dados no *Amazon S3*. Cada cluster *Spark* foi formado a partir de um nó *Master* (*Driver Program*) conectado a um ou mais nós *Workers* (cujo número, é denotado por **# W**). Para cada *Worker*, foi definido apenas um *Spark Executor*. Para cada configuração de *cluster Spark*, a quantidade de *Map Tasks* (**# Map**) foi definida conforme a quantidade total de *vCPUs* disponíveis no respectivo *cluster*, enquanto que a quantidade de *Reduce Tasks* (**# Red**) foi variada considerando todos os possíveis divisores de **# Map**, *i.e.*, **# Red** = $\{r \in \mathbb{N}^* \mid r \setminus \textbf{\# Map}\}$. Para cada cenário formado, os volumes de dados de entrada (I) considerados foram 2 *GB* e 4 *GB*. No total, 174 cenários experimentais foram formados.

A **Tabela 2** apresenta os tempos e custos médios considerando cinco execuções por cenário, enfatizando as combinações de **# W**, **# Map**, e **# Red** que obtiveram os

[11] https://aws.amazon.com/pt/cloudwatch/

[12] https://aws.amazon.com/pt/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/

[13] https://aws.amazon.com/pt/blogs/aws/now-available-r5-r5dand-z1d-instances/

menores tempos de execução da aplicação. Note que, a combinação ⟨**# W** = 13, **# Map** = 26, **# Red** = 26⟩ obteve o melhor custo-benefício para 2 *GB* de entrada, com custo monetário de 0,01 *USD* e tempo de execução de 71,29 s. Para 4 *GB*, o melhor custo-benefício foi atingido com ⟨**# W** = 16, **# Map** = 32, **# Red** = 32⟩, com custo monetário de 0,02 *USD* e tempo de execução de 97,27s.

Tabela 2. Tempos e custos médios de execução do *Word-Count-Spark*.

Quantidade de Workers (# W)	Input (I): 2 GB				Input (I): 4 GB			
	# Map	# Red	Tempo (s)	Custo (USD)	# Map	# Red	Tempo (s)	Custo (USD)
1	2	2	362,70	0,01	2	2	693,45	0,02
2	4	4	274,70	0,01	4	4	364,03	0,01
3	6	6	145,68	0,01	6	6	253,98	0,01
4	8	8	119,55	0,01	8	8	201,89	0,01
5	10	10	107,09	0,01	10	10	170,75	0,01
6	12	12	95,07	0,01	12	12	150,72	0,01
7	14	14	87,73	0,01	14	14	141,47	0,02
8	16	16	85,20	0,01	16	16	130,01	0,02
9	18	18	79,13	0,01	18	18	117,98	0,02
10	20	20	81,18	0,01	20	20	111,64	0,02
11	22	22	75,54	0,01	22	22	111,24	0,02
12	24	24	76,21	0,01	24	24	102,89	0,02
13	26	26	71,29	0,01	26	26	104,27	0,02
14	28	28	71,58	0,01	28	28	101,93	0,02
15	30	30	74,28	0,02	30	30	98,08	0,02
16	32	32	75,93	0,02	32	32	97,27	0,02
32	64	32	73,77	0,03	64	64	98,05	0,04

5.3. Comparando as versões MARLA e Spark

Atualmente, a capacidade máxima de memória disponível por função Lambda na *AWS Lambda* é de 10.240 MB[12] (10 *GB*). Esta restrição motivou o uso da instância *r5.large* para a construção do cluster *Spark* na *AWS EC2*, visto que é a instância *memory optimized* com capacidade de memória (16 *GiB*) mais próxima daquela. Quanto à capacidade de processamento, não foi possível determinar configurações que fossem totalmente equivalentes. Na *AWS EC2*, o cliente é capaz de obter a informação da quantidade de *vCPUs* disponíveis por instância de um modo transparente (em particular, 2 *vCPUs* para a *r5.large*). Conhecimento similar já não é possível no *AWS Lambda*, pois esta informação não se encontra disponibilizada. Sabe-se apenas que, dependendo da quantidade de memória alocada, são oferecidas de 2 a 6 *vCPUs* por função Lambda.

Para a análise comparativa de custos computacionais utilizando *FaaS* (*MARLA*) e *IaaS* (*Spark*), selecionamos os melhores tempos de execução da aplicação *Word-Count*, considerando os volume de dados de entrada de 2 *GB* e 4 *GB*. A **Tabela 3** sumariza tais resultados. Com o *MARLA* na *AWS Lambda* foi possível executar a aplicação mais rapidamente: aproximadamente 31,2% para a entrada de 2 *GB* e 6,6% para a entrada de 4 *GB*. Em contrapartida, com o *Spark* na *AWS EC2* foi possível executar a aplicação de modo mais econômico: aproximadamente 87,5% para a entrada de 2 *GB* e 86,6% para a entrada de 4 *GB*.

Alguns fatores não foram considerados nesta análise, tais como: o tempo entre a chamada da função lambda e sua efetiva execução (*cold start*), tratamento de recursos ociosos na *AWS EC2*, estouro do tempo de execução, e utilização mais eficiente do *Amazon S3*. Do ponto de vista do cluster *Spark*, deve-se evitar ao máximo a subutilização de recursos, por exemplo, por falta de trabalho a ser executado. É preciso avaliar as vantagens

Tabela 3. MARLA Vs. Spark: melhores tempos médios de execução.

Ambiente	Quantidade de Mappers	Quantidade de Reducers	Lambda Function Memory (MB)	Input (GB)	Tempo (s)	Custo (USD)
MARLA + AWS Lambda	64	32	2048	2	49,07	0,08
				4	90,83	0,15
Spark + AWS EC2	26	26	N/A	2	71,28	0,01
	32	32		4	97,27	0,02

e desvantagens em manter um *cluster* configurado, ainda que os clientes sejam cobrados apenas pelos volumes de armazenamento *EBS* e de endereço *IP* público reservado que estejam associados a cada instância interrompida (neste caso as instâncias de máquinas virtuais do cluster ficariam em estado *Stopped*)[14]. Do ponto de vista do *MARLA*, deve-se evitar ao máximo o estouro do tempo de execução permitido para cada função Lambda, atualmente limitado a quinze minutos. Um tratamento adequado à aplicação pode ser requerido, dependendo da carga de trabalho a ser processada no *AWS Lambda*. Além disso, é preciso analisar as vantagens e desvantagens de realizar leituras e escritas de dados parciais no *Amazon S3* durante o processamento do *MapReduce* com *MARLA*, pois estas ações também possuem custos monetários associados.

6. Conclusões e Direções Futuras

Esse trabalho oferece uma análise comparativa entre as execuções de uma aplicação *MapReduce* sob a ótica dos frameworks *MARLA* e *Spark* na AWS, não disponibilizado na literatura correlata. A análise realizada neste trabalho indica que a utilização do modelo *FaaS* para a execução de aplicações *MapReduce* através do framework *MARLA* pode ser vantajosa, visto que o usuário somente especifica a lógica das funções *Map* e *Reduce*, enquanto que o framework se encarrega de gerar a infraestrutura necessária para a execução da aplicação. Já na abordagem com *Spark* no modelo *IaaS*, além da especificação das funções *Map* e *Reduce*, faz-se necessária a implantação manual do cluster *Spark* com a respectiva especificação de *Workers* e *Executors*, conforme descrito na Subseção 4.2. Em termos de codificação da aplicação, as duas abordagens possuem dificuldades similares, uma vez que será necessária a escolha da estrutura de dados e a aplicação das operações adequadas para a geração e o processamento dos pares de chave-valor utilizados na abordagem MapReduce. A avaliação experimental da aplicação *Word-Count* apontou benefícios no desempenho de execução para a versão *MARLA* quando comparada à versão *Spark*, em detrimento de um maior custo monetário.

Apesar do *FaaS* propor um desacoplamento do usuário em relação à infraestrutura e à execução, ainda é necessário especificar algumas características do modelo *MapReduce*, tais como o número de *Mappers* e *Reducers*. Além disso, a definição da quantidade de memória alocada por função Lambda afeta diretamente o custo monetário. A versão implementada com *Spark* foi executada usando apenas um tipo de instância de máquina virtual no *AWS EC2*, cujos recursos computacionais foram similares aos recursos especificados para as funções Lambda no *MARLA*. A seleção de outros tipos de instâncias, bem como o dimensionamento e o balanceamento de carga, são etapas que devem ser consideradas e que podem melhorar o desempenho (trabalho em andamento).

[14] `https://aws.amazon.com/premiumsupport/knowledge-center/ec2-billing-terminated/`

Como trabalho futuro, pretende-se elaborar um modelo de otimização da configuração do ambiente *MARLA* na *AWS Lambda* considerando uma certa aplicação, visando a redução de tempo e custo monetário associados à sua execução. Além disso, avaliar benchmarks de *Big Data*, variando os dados de entrada e realizar comparações de desempenho do *MARLA* com *frameworks* similares para *FaaS*, por exemplo, o *Flint* [Kim and Lin 2018].

Referências

[Awaysheh et al. 2021] Awaysheh, F. M., Alazab, M., Garg, S., Niyato, D., and Verikoukis, C. (2021). Big data resource management & networks: Taxonomy, survey, and future directions. *IEEE Communications Surveys & Tutorials*, 23(4):2098–2130.

[Dean and Ghemawat 2004] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI' 04, pages 137–149. USENIX Association.

[Giménez-Alventosa et al. 2019] Giménez-Alventosa, V., Moltó, G., and Caballer, M. (2019). A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems*, 97:259–274.

[Kapil et al. 2022] Kapil, D., Mishra, S., and Gupta, V. (2022). A performance perspective of live migration of virtual machine in cloud data center with future directions. *International Journal of Wireless and Microwave Technologies*, 12:48–56.

[Kim and Lin 2018] Kim, Y. and Lin, J. (2018). Serverless Data Analytics with Flint. In *IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, Los Alamitos, CA, USA. IEEE Computer Society.

[Malawski et al. 2020] Malawski, M., Gajek, A., Zima, A., Balis, B., and Figiela, K. (2020). Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 110:502–514.

[Muniswamaiah et al. 2019] Muniswamaiah, M., Agerwala, T., and Tappert, C. (2019). Big data in cloud computing review and opportunities. *International Journal of Computer Science & Information Technology (IJCSIT)*, 11.

[Nunes et al. 2021] Nunes, A. L., Melo, A., Boeres, C., de Oliveira, D., and Drummond, L. M. A. (2021). Towards Analyzing Computational Costs of Spark for SARS-CoV-2 Sequences Comparisons on a Commercial Cloud. *XXII Symposium in High Performance Computing Systems*, pages 192–203.

[Shvachko et al. 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 15–28, USA. USENIX Association.