UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ROBERTA DE SOUSA MOREIRA

AVALIAÇÃO DAS LINGUAGENS PYTHON E GO NA PLATAFORMA AWS LAMBDA PARA APLICAÇÕES DE COMPUTAÇÃO DE ALTO DESEMPENHO

QUIXADÁ
2021

ROBERTA DE SOUSA MOREIRA

AVALIAÇÃO DAS LINGUAGENS PYTHON E GO NA PLATAFORMA AWS LAMBDA PARA APLICAÇÕES DE COMPUTAÇÃO DE ALTO DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação, como parte dos requisitos necessários à obtenção do título de Bacharel pela Universidade Federal do Ceará.

Orientador: Prof. Dr. João Marcelo Uchôa de Alencar

QUIXADÁ

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M839a Moreira, Roberta de Sousa.
Avaliação das linguagens python e go na plataforma AWS lambda para aplicações de computação de alto desempenho / Roberta de Sousa Moreira. – 2021.
45 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2021.
Orientação: Prof. Dr. João Marcelo Uchôa de Alencar.

1. Computação de alto desempenho. 2. Serviços da Web. 3. Python (Linguagem de programação de computador). I. Título.

CDD 004

ROBERTA DE SOUSA MOREIRA

AVALIAÇÃO DAS LINGUAGENS PYTHON E GO NA PLATAFORMA AWS LAMBDA PARA APLICAÇÕES DE COMPUTAÇÃO DE ALTO DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação, como parte dos requisitos necessários à obtenção do título de Bacharel pela Universidade Federal do Ceará.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____
Prof. Dr. João Marcelo Uchôa de Alencar (Orientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Emanuel Ferreira Coutinho
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Jefferson de Carvalho Silva
Universidade Federal do Ceará (UFC)

RESUMO

Com o aumento da utilização do paradigma *serverless* para utilização de aplicações de alto desempenho por conta do ambiente propício que a nuvem proporciona através de sua escalabilidade e elasticidade, surgem alguns questionamentos iniciais ao desenvolvedor que pretende executar sua aplicação de alto desempenho em uma arquitetura *serverless*. Com isso, conhecer os benefícios e limitações dessa arquitetura torna-se algo relevante para estes desenvolvedores. Um dos questionamentos iniciais que surgem é a respeito da melhor linguagem a ser utilizada para a construção destas aplicações. Buscando solucionar estes questionamentos iniciais, este trabalho aborda uma análise das linguagens *Python* e *Go* utilizadas na construção de aplicações de alto desempenho no paradigma serverless na plataforma da nuvem *Amazon Web Services* (AWS), realizando a execução de funções, no serviço AWS *Lambda*, de Multiplicação de Matrizes Serial, problema bastante utilizado em diversos trabalhos. Com esses experimentos foi possível constatar que a melhor alternativa dentre as definidas é o *Python* com a utilização da biblioteca *NumPy*. E entre *Go* e *Python* puras, ou seja, sem utilização de bibliotecas a melhor opção é a linguagem *Go*.

Palavras-chave: Computação de alto desempenho. Serviços da Web. Python (Linguagem de programação de computador).

ABSTRACT

With the use of the serverless paradigm to use high performance applications due to the enabling environment of a cloud through its scalability and elasticity, some initial questions arise to the developer who intends to run his high performance application in a serverless architecture. Thus, knowing the benefits and limitations of this architecture becomes relevant for these developers. One of the initial questions that arise is about the best language to be used to build these applications. Seeking to solve these initial questions, this work addresses an analysis of the Python and Go languages used in the construction of high performance applications in the serverless paradigm on the Amazon Web Services (AWS) cloud platform, performing a function execution, in the AWS Lambda service, of Multiplication of Serial Matrices, a problem widely used in several works. With these possible experiments, verify that the best alternative choice as defined is Python using the NumPy library. And between Go and Python pure, that is, without using libraries, the best option is the Go language.

Keywords: High Performance Computing. Web Service. Python (Computer programming language).

SUMÁRIO

1 INTRODUÇÃO 7

1.1 OBJETIVOS 9

1.1.1 Objetivo Geral 9

1.1.2 Objetivos Específicos 9

1.2 Organização 10

2 TRABALHOS RELACIONADOS 10

2.1 Trabalho 01: *Serverless Big Data Processing using Matrix Multiplication as Example* 10

2.2 Trabalho 02: *numpywren: Serverless Linear Algebra* 12

2.3 Trabalho 03: *FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC.* 14

2.4 Trabalho 04: *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions.* 16

2.5 Avaliação 18

3 FUNDAMENTAÇÃO TEÓRICA 19

3.1 Computação de Alto de Desempenho 19

3.1.1 Modelos de Programação 20

3.1.2 Granularidade de Aplicações 21

3.2 Computação em Nuvem 22

3.2.1 Vantagens e Motivação 23

3.2.2 Modelos de Implantação 24

3.2.3 Modelos de Serviços 25

3.3 Computação Sem Servidor 26

3.3.1 Fundamentos de Computação Sem Servidor 26

3.3.2 Benefícios da Computação Sem Servidor 27

3.3.3 Desafios para Computação Sem Servidor 27

3.3.4 AWS Lambda 28

3.3.5 Exemplo de Arquitetura de Aplicação Computação Sem Servidor 32

4 PROCEDIMENTOS METODOLÓGICOS 34

4.1 Etapas de experimento 34

4.1.1 Definição do Formato das Matrizes 34
4.1.2 Implementação de Funções de Multiplicação 35
4.1.3 Submissão de um Conjunto de Matrizes 36
5 RESULTADOS 37
6 CONCLUSÃO 41
REFERÊNCIAS 42

1 INTRODUÇÃO

A Computação Sem Servidor (*serverless*) é uma arquitetura nativa da nuvem que transfere a responsabilidade das questões operacionais com infraestrutura para a plataforma gerenciadora de serviços na nuvem, como a *Amazon Web Service* (AWS), *Google Cloud Platform* ou *Microsoft Azure*. Essa arquitetura vem sendo amplamente utilizada para Computação de Alto Desempenho, ou *High-Performance Computing* (HPC), por possuir infraestrutura elástica e escalável que proporciona um ambiente propicio para execução de aplicações paralelas. Portanto, conhecer capacidades e limitações dessa arquitetura se torna necessário para quem deseja rodar aplicações HPC na arquitetura *Serverless*. De acordo Sterling, Anderson e Brodowicz:

> Computação de Alto Desempenho é uma área de esforços que se relacionam com todas as facetas tecnológicas, metodológicas e de aplicação associadas com alcançar a maior capacidade computacional possível em qualquer instante no tempo e nível de desenvolvimento tecnológico. Ela faz uso de uma classe de máquinas eletrônicas digitais chamadas de "supercomputadores" para resolver uma vasta gama de problemas computacionais ou "aplicações" (também chamadas de "cargas de trabalho") da maneira mais rápida quanto possível. A execução de uma aplicação em um supercomputador é chamada de "supercomputação", sendo sinônimo de Computação de Alto Desempenho. (2018, p. 3, tradução nossa)

A Computação de Alto Desempenho está presente em inúmeras áreas, em aplicações "[...] tradicionais, como genômica, química computacional, modelagem de risco financeiro, engenharia auxiliada por computador, previsões climáticas e imagens sísmicas, assim como aplicativos emergentes, como *machine learning*, aprendizagem profunda e direção autônoma." (AWS, 2020a). Em comum a todas essas soluções, temos a utilização de cálculos numéricos em álgebra linear.

As aplicações de Computação de Alto Desempenho em geral necessitam de grandes recursos computacionais, para isso, são utilizados *clusters* de vários computadores que compõe uma infraestrutura, local ou em nuvem, mais abrangente que se adeque aos recursos necessários para essa execução. Infraestruturas locais geralmente necessitam de um conhecimento amplo por parte do desenvolvedor, ou engenheiro, o que implica em uma dedicação maior para a realização de configurações específicas. Possuir o entendimento sobre toda uma arquitetura local de computadores exige um tempo a mais de desenvolvimento para uma determinada aplicação, além disso, pode gerar custos mais altos e recursos desnecessários, pois não possui elasticidade como em uma infraestrutura na nuvem.

Oferecendo um conjunto de recursos amplos e com configurações flexíveis, a nuvem disponibiliza infraestrutura e plataformas que permitem a utilização de serviços que auxiliam e facilitam a criação de *clusters* virtuais em relação a manutenção de infraestrutura local. Isso permite que pesquisadores possam utilizar computação em nuvem para executar aplicações de computação de alto desempenho. Para isso, o pesquisador, ou desenvolvedor, não precisará montar a infraestrutura física, mas ainda assim, precisará entender a configuração do *software* e sistema operacional do *cluster* virtual.

Como dito anteriormente, com o paradigma *Serverless*, as plataformas na nuvem gerenciam e disponibilizam infraestruturas adequadas que permitem a pesquisadores, desenvolvedores, engenheiros e proprietários de sistemas HPC a utilizar-se desses recursos sem as limitações que uma infraestrutura local poderia vir a trazer, e diferente da computação em nuvem, não necessitará de conhecimento detalhado sobre as configurações de infraestrutura e *software*. A computação *Serverless* possibilita a vantagem de, em teoria, preocupar-se apenas com o código da aplicação encapsulado em funções, a nuvem trata de executar esse código e o custo é equivalente apenas as funções executadas. A vantagem seria facilidade de uso com custo inferior ou igual de uma abordagem local.

É importante analisarmos também restrições, como exemplo, a coordenação das funções na nuvem, pois o nível de limitação dessa restrição dependerá da aplicação em questão, mas ainda assim, a nuvem oferece serviços que auxiliam nesse quesito. Outra restrição é o número restrito de linguagens disponíveis para utilização nas funções na nuvem, geralmente são linguagens bem conhecidas entre a comunidade e com bom suporte, mas pode vir a ser um problema para uma aplicação específica ou código legado. O acesso aos dados, ao armazenamento, não é direto, como o acesso a um arquivo em um diretório, é preciso utilizar os serviços de armazenamento da nuvem, essa restrição pode limitar uma aplicação em quesitos de agilidade e flexibilidade, em relação ao armazenamento local.

Dentre as restrições levantadas, a questão de qual linguagem utilizar para criar as funções de uma aplicação *Serverless* é uma das primeiras a ser encarada pelo desenvolvedor. Duas opções bastante populares é a ferramenta *NodeJS* e a linguagem *Python*. A primeira (CHANIOTIS, KYRIAKOU e TSELIKAS, 2015) tem no seu projeto a preocupação com o desempenho de aplicações *web*, que não apresentam os mesmos requisitos de aplicações HPC. Já *Python* tem sido usada com sucesso para HPC (OLIPHANT, 2007), mas é uma linguagem interpretada. Uma das opções que permitem a execução de código compilado é a linguagem Go (MEYERSON, 2014), em geral as linguagens com opção de compilação apresentam melhor

desempenho. Entretanto, não há estudos comparando o desempenho entre Python e Go em uma arquitetura *Serverless*, queremos confirmar esse cenário na nuvem em funções AWS Lambda.

Considerando que o contexto das limitações apresentadas, o objetivo desse trabalho é contribuir com investigações a respeito das linguagens disponíveis, buscando responder qual seria a linguagem com melhor performance para computação de alto desempenho no paradigma *serverless* na plataforma de nuvem pública AWS, apresentando nossos resultados para nosso público-alvo que são os desenvolvedores interessados em executar aplicações paralelas na nuvem.

1.1 OBJETIVOS

Nesta seção apresentaremos o objetivo geral deste trabalho, assim como, os objetivos específicos.

1.1.1 Objetivo Geral

Este trabalho tem como finalidade determinar, entre as linguagens *Go* e *Python,* a melhor opção para execução de aplicações de alto desempenho no paradigma *Serverless* na plataforma AWS.

1.1.2 Objetivos Específicos

A partir dos objetivos específicos abaixo, concluiremos o objetivo geral apresentado.

1. Averiguar a viabilidade da execução de aplicações de alto desempenho na linguagem Python na AWS *Lambda*.
2. Averiguar a viabilidade da execução de aplicações de alto desempenho na linguagem Go na AWS *Lambda*
3. Desenvolver aplicações de alto desempenho nas linguagens *Python* e *Go* para resolver um mesmo problema na AWS *Lambda.*
4. Delimitar um conjunto de dados de entrada para submeter para ambas as aplicações em execução na AWS *Lambda*.
5. Medir o tempo de execução partindo dos dados de entrada nas aplicações *Go* e *Python*.

1.2 Organização

O restante deste trabalho é organizado da seguinte forma: no Capítulo 1 apresentamos a introdução e objetivos de nosso trabalho. O Capítulo 2, apresenta todos os trabalhos relacionados que fizeram parte de nossas pesquisas bibliográficas, além de indicar uma avaliação a respeito destes. No Capítulo 3, nossa fundamentação teórica é indicada com as principais definições que motivam este trabalho. Capítulo 4 apresenta as etapas de nosso procedimento metodológico que indica o planejamento de construção e conclusão de nosso objetivo geral e no Capítulo 5 apresentamos nossos resultados obtidos com a realização das etapas descritas no capítulo anterior e a conclusão partindo desses resultados.

2 TRABALHOS RELACIONADOS

Nesta seção serão apresentados os principais artigos que serviram como base para este trabalho.

2.1 Trabalho 01: *Serverless Big Data Processing using Matrix Multiplication as Example*

Em (WERNER, KUHLENKAMP, *et al.*, 2018) os autores exploram a área de *Big Data* com o exemplo de multiplicação de matrizes, no qual foram definidos requisitos para o projeto de *Big Data* em Computação Sem Servidor, além do desenvolvimento de um protótipo de multiplicação de matrizes usando Funções como Serviço (FaaS). Partindo desse contexto os autores apresentam seus resultados através de experimentos, levantando-se direções para o projeto de futuros aplicativos de *Big Data Serverless*.

Os autores apresentaram requisitos não funcionais genéricos para orientar o projeto de aplicativos de *Big Data Serverless*: escalabilidade, que se diz respeito à capacidade do sistema de realizar multiplicações de matrizes maiores de maneira eficaz e adaptar os recursos utilizados para oferecer melhor desempenho para matrizes de um tamanho fixo; gerência automática, pois o sistema terá de se utilizar de serviços na nuvem totalmente gerenciados não necessitando da intervenção de um desenvolvedor; desempenho, o sistema deverá fornecer desempenho melhor ou semelhante ao de soluções baseadas em máquinas virtuais; custo, que devem ser mais baixos ou semelhantes aos custos de soluções baseadas em máquinas virtuais de estruturas de computações distribuídas; e adaptabilidade, sistemas distribuídos escaláveis permitem adicionar ou remover recursos de infraestrutura de nuvem em uma implantação (i) para aumentar o desempenho e os custos de infraestrutura ou (ii) diminuir o desempenho e os

custos de infraestrutura, ou seja o sistema deve permitir que o desenvolvedor ajuste essa troca de acordo com os requisitos de aplicação.

O projeto do sistema foi desenvolvido como uma aplicação de multiplicação de matrizes. Um desafio para os autores foi adaptar a aplicação para o paradigma sem servidor, no qual a computação com estados é limitada, tendo que usar um serviço separado para coordenação de funções, além de projetar uma aplicação que seja ajustável em relação ao custo e desempenho. O projeto de sistema é separado entre três elementos principais: Cálculo, Armazenamento e Orquestração.

O componente Cálculo é tratado em funções sem estados como o paradigma exige, as funções desse componente obtêm a entrada e armazenam a saída em serviços na nuvem. O componente Orquestração garante a distribuição da carga de trabalho entre as funções do Cálculo. Cada computação realizada pelo componente passa pelo carregamento de partições necessárias de fatores ou resultados intermediários do armazenamento, realização dos cálculos designados e escrita dos resultados de volta no armazenamento. Esse projeto geral garante que o sistema seja compatível com o paradigma sem servidor.

O componente Cálculo define uma arquitetura básica de funções fixas, nas quais se multiplicam matrizes quadráticas distribuídas. Essa arquitetura básica pode ser replicada várias vezes e executada em paralelo para abranger mais intervalos de matrizes e permitir cálculos maiores. O algoritmo de Strassen foi utilizado para facilitar o cálculo distribuído, pois permite distribuir uma multiplicação entre sete funções independentes, cada uma lendo sua matriz de entrada do armazenamento e gravando os resultados. Logo em seguida inicia-se o estágio do coletor, os valores intermediários são carregados do armazenamento e os valores finais são gravados novamente.

Um dos desafios encontrados é identificar quando todos os cálculos intermediários foram executados, pois na computação sem servidor não existe o conceito de estados sem que seja necessário o uso de um serviço específico de orquestração. O conjunto de funções que realizam a multiplicação é orquestrado através do conceito de Unidade de Computação de Strassen (UCS).

A UCS básica utiliza de sete instâncias para executar a paralelização de multiplicação de matrizes. A coordenação utilizada da UCS é a partir de um mecanismo de máquinas de estados fornecidos pela AWS (*Step Functions*), esse mecanismo garante que a função de coletor só seja acionada após a resolução de todos os subproblemas, além de garantir a escalabilidade da aplicação. Após a distribuição de metadados necessários para as instâncias

de funções individuais, feito pelo componente de Orquestração, ocorre o cálculo e depois os resultados podem ser consolidados na pasta de armazenamento.

Esse projeto de sistema usa o AWS S3 para armazenamento, AWS *Lambda* para computação e AWS *Step Functions* para orquestração. As funções *Lambdas* foram implementadas usando *Python*, os autores justificam o uso por suas boas bibliotecas para manipular os dados de matrizes e executar multiplicação local. O AWS *Step Functions* utilizado na orquestração da aplicação é um serviço criado em 2016 pela Amazon, que resolve os problemas que surgiram no AWS *Lambda* pela falta de computação com estados no paradigma sem servidor. Foram realizados três experimentos para avaliar o quão bem o sistema atendeu aos requisitos de projeto dado pelos autores e o quanto se compara aos sistemas distribuídos oferecidos na nuvem totalmente gerenciados. Avaliou-se a escalabilidade e a capacidade de adaptação do sistema projetado, comparando os custos de desempenho e trabalho das implementações dos clusters *Apache Spark* e *Hadoop MapReduce* gerenciados pelo Amazon EMR.

Foram comparadas estratégias de dimensionamento baseadas em divisões e unidades nas dimensões e partições das matrizes de entrada, ambas aumentando o paralelismo. Os experimentos indicam que o parâmetro de divisão, que orientam a quantidade de trabalho para uma função acumuladora, é um ponto de ajuste eficaz para configurar a ajustabilidade existente no projeto do sistema. Portanto, definir configurações apropriadas para limites de função e tamanho de divisão são duas tarefas de gerenciamento restantes na versão atual do sistema.

Este trabalho trata-se de um exemplo de aplicação paralela em Computação sem Servidor, os autores só usam uma linguagem, mas este trabalho é importante para o nosso porque fornece uma aplicação que podemos usar como estudo de caso. A partir desse contexto os autores conseguem chegar à conclusão que o processamento de *Big Data Serverless* pode competir, corresponder e superar as soluções distribuídas baseadas em *cluster*, em termos de desempenho, escala e custos. Com isso, a Computação Sem Servidor deve ser considerada uma nova alternativa relevante em relação a processamento de *Big Data.*

2.2 Trabalho 02: *numpywren: Serverless Linear Algebra*

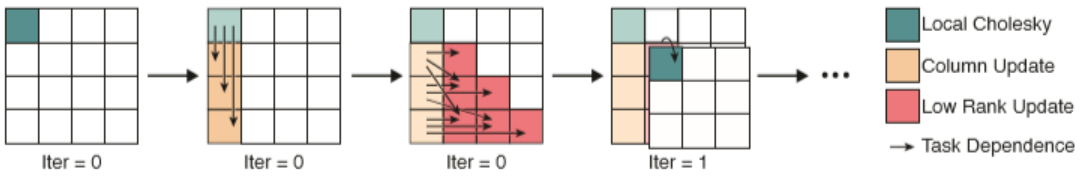
A Computação Sem Servidor abrange diversas aplicações dentro da computação científica, tais como operações de álgebra linear. Para que os cálculos algébricos possam ser executados com o mesmo desempenho da maneira tradicional, com acesso a *clusters* de supercomputadores, é necessário lidar com alguns desafios que surgem no ambiente sem

servidor. Abordaremos o assunto através do trabalho de Shankar et al. (2018) que com a ferramenta *numpywren* demonstram como as plataformas sem servidor podem ser eficientes para esse tipo de aplicação.

Numpywren surgiu através da observação de que para muitas operações lineares o tempo de computação geralmente domina o tempo de comunicação, e que o uso de *pipeline* e devidos bloqueios possibilita usar armazenamento distribuído como substituto para a memória compartilhada. O *numpywren* surge como um sistema para álgebra linear em arquiteturas sem servidor que executa os diversos cálculos em funções sem estado enquanto armazena estados intermediários em armazenamento distribuído. Além disso, a aplicação executa programas escritos em *LAmbdaPACK*, uma *DSL* que facilita a coordenação da comunicação durante os cálculos.

Os autores projetaram o *numpywren* para direcionar cargas de trabalho de álgebra linear que possuem padrões de execução semelhante ao algoritmo de decomposição de Cholesky, uma solução que evita a comunicação e é muito usado na resolução de equações lineares. Neste algoritmo há a divisão da matriz em, minimizando a transferência total de dados. A decomposição de Cholesky apresenta paralelismo dinâmico e que possui dependências refinadas entre iterações e dentro de uma iteração. A decomposição utilizada pelos autores pode ser vista abaixo:

Figura 1 - Representação da Decomposição de Cholesky.



Fonte:(SHANKAR ET AL., 2018).

Legenda: Primeiros 4 passos de decomposição paralela de Cholesky: 0) Decomposição de Cholesky do bloco diagonal; 1) Atualização da coluna paralela; 2) Atualização da submatriz paralela; 3) Decomposição de Cholesky do bloco diagonal (subsequente). Esta decomposição revela o paralelismo dinâmico do algoritmo.

O objetivo da aplicação desenvolvida por Shankar et al. (2018) é poder se adaptar a quantidade de paralelismo, abordando isso através de decomposições de programas em unidades de execução refinadas que são executadas em paralelo. Para que isso ocorra no ambiente sem servidor, os autores propõem a descentralização da análise de dependência. Um grafo global de dependência descrito utilizando a linguagem DSL *LAmbdaPack* é distribuído,

descrevendo o fluxo de programa para todos os trabalhadores, assim cada trabalhador raciocina localmente sobre suas dependências com base em sua localização atual no grafo.

A avaliação do *numpywren* ocorreu através de quatro algoritmos, que possuem complexidade $O(n^3)$ e diferem em padrões de acessos a dados, de álgebra linear: a Multiplicação de Matriz (GEMM), Decomposição de QR (QR), Decomposição em Valores Singulares (SVD) e Decomposição de Cholesky. Para os quatro algoritmos, os autores compararam o *ScaLAPACK*, uma biblioteca *Fortran* para álgebra linear distribuída de alto desempenho. Além de realizar uma análise detalhada de escalabilidade e tolerância a falhas do *numpywren* usando o Cholesky e comparando esse desempenho ao *Dask*, uma biblioteca tolerante a falhas baseada em *python* que suporta álgebra linear distribuída.

O *numpywren* foi implementado em *Python* na plataforma *Amazon Web Service* (AWS), para o armazenamento de estados usou-se o *Redis*, que é um armazenamento de valor-chave do *ElasticCache* que não é "sem servidor", também foi usado o serviço de filas simples *Amazon SQS*, para filas de tarefas, o *Lambda* para a execução de funções e o *S3* para o armazenamento de objetos. O *ScaLAPACK* e o *Dask* foram executados em instâncias *c4.8xlarge*.

Os autores concluem que uma das principais desvantagens do modelo sem servidor é a alta comunicação necessária devido à falta de localidade e primitivas de transmissão eficientes. Uma maneira de aliviar isso seria execuções sem servidor mais robustas (por exemplo, 8 *cores* em vez de 1) que processam partes maiores dos dados de entrada, além disso, otimizações como *pipeline* melhoram o perfil de utilização de recursos. Com isso, o sistema proposto pelos autores, *numpywren*, pode ser usado para programas intensivos em computação com complexas rotinas de comunicação oferecendo facilidade de uso e tolerância direta a falhas, através da análise da linguagem *LAmbdaPACK* intermediária, permitindo adaptação dinâmica ao paralelismo inerente aos algoritmos de álgebra linear com a elasticidade fornecida pela computação sem servidor.

Assim como o trabalho 01, o trabalho 02 reforça a importância da Álgebra Linear para computação paralela, mas também só considera a linguagem Python. O nosso trabalho pretende avaliar também uma linguagem compilada, Go.

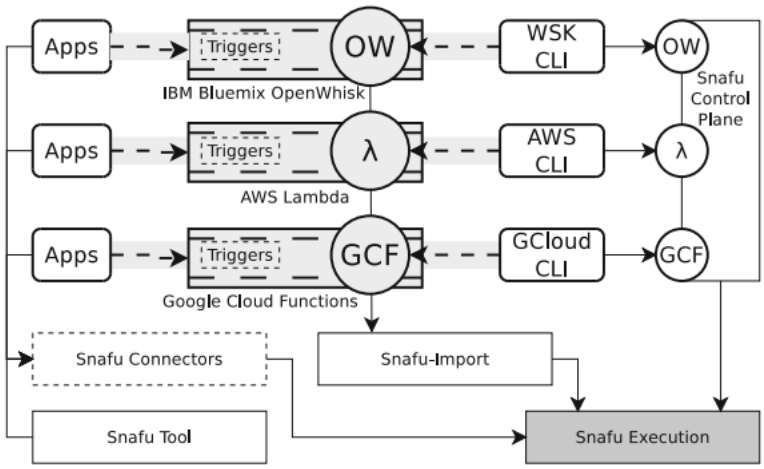
2.3 Trabalho 03: *FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC.*

Para que aplicações de alto desempenho possam executar em plataformas na nuvem, é necessário configurações genéricas requisitadas pelo domínio da aplicação. Spillner,

Mateos e Monge (2017) abordaram em seu artigo a adaptação de *softwares* para o *FaaS* através de um processo chamado *FaaSification.*

Os autores conduziram quatro experimentos, em *Python*, para comparar o desempenho e outras características relacionadas a recursos no *FaaS.* Os experimentos foram realizados com alguns fornecedores comerciais de FaaS, bem como com a ferramenta FaaS de código aberto Snafu, que permite gerenciar, executar e testar funções através de interfaces específicas do fornecedor como mostra a Figura 2. Foi escolhido quatro domínios de aplicação: matemática (cálculo de π), computação gráfica (detecção de rosto), criptografia (quebra de senha) e meteorologia (previsão de precipitação). Os três primeiros experimentos são sintéticos, enquanto o quarto usou *FaaSification* para analisar um aplicativo não-FaaS existente.

Figura 2 - Ecossistema do Snafu e suas ferramentas.



Fonte: (SPILLNER; MATEOS; MONGE, 2018).

O primeiro experimento foi desenvolvido em *Python3* e *Python2,* em ambas as versões do *Python* a execução em *FaaS* ocorreu de maneira mais rápida. No segundo experimento em *Python2,* o desempenho da versão *serverless* fica significativamente baixo, devido a centralização de E/S da função, e só se torna eficaz quando usadas 19 ou mais *threads.* No terceiro, a execução *serverless* também teve uma execução mais rápida. O quarto experimento, no qual foi utilizado *FaaSification* foi necessário ferramentas para transformar o código existente em funções com conformidades com as convenções da plataforma provedora na nuvem, já que se trata de um aplicativo não-FaaS existente. Esse processo de transformação da aplicação é chamado de *FaaSification,* nesse experimento são utilizadas várias funções monolíticas, que precisam ser subdivididas em outras funções, para que não se exceda os limites de tempo de execução dos provedores de *FaaS.*

Com os experimentos os autores mostram que em muitos domínios da computação científica e de alto desempenho, as soluções podem ser baseadas em funções simples executadas em plataformas *FaaS*. E para que essas aplicações possam ser executadas com sucesso é necessário o uso de ferramentas que nos auxiliem com o processo de engenharia, tendo em vista que aplicações de alto desempenho geralmente necessitarão de configurações especiais.

Novamente temos um trabalho que como o 1 e 2 só considera Python, mas se testou outras soluções de nuvem além da AWS. A princípio, vamos ficar restritos na AWS, mas este trabalho é importante para mostrar que no futuro podemos avaliar outras nuvens.

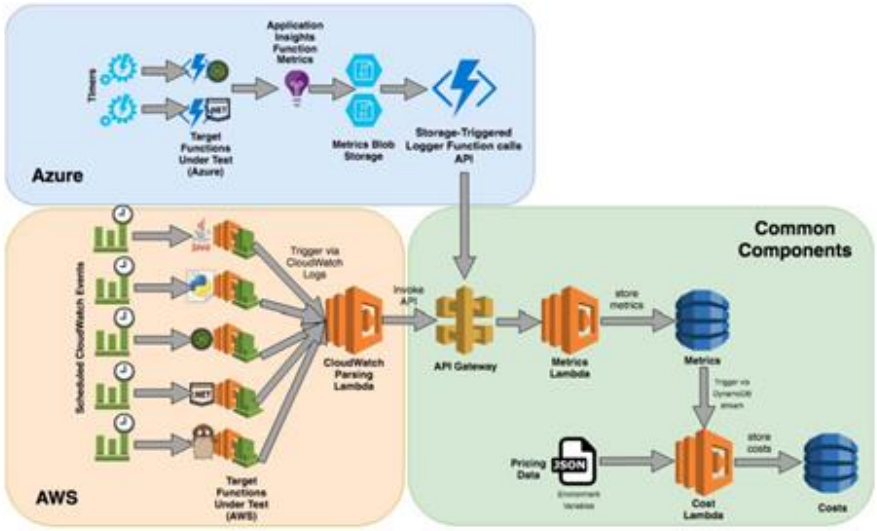
2.4 Trabalho 04: *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions.*

A escolha da linguagem de programação a ser usada nas funções da nuvem influencia diretamente no desempenho e custos da execução. O trabalho de Jackson e Clynch (2018) analisa linguagens disponíveis nas plataformas AWS *Lambda* e *Azure Functions*. Essa análise é realizada através de um modelo *CostHat* que aplica resultados de teste de desempenho e custo em uma arquitetura realista sem servidor. As cinco linguagens escolhidas para que seus tempos de execução possam ser medidos na AWS *Lambda* foram: *NodeJS, Go, Python, Java* e *.NET Core 2*. Já no *Azure Functions* foram o *NodeJS* e *.NET C#.* Os autores projetaram uma série de testes de *Warm-Start*, que é quando uma plataforma sem servidor reutiliza um contêiner de execução existente em vez de criar um ambiente novo para executar uma função; e *Cold-Start*, que ocorre quando não há contêiner disponível para reutilização, portanto, um contêiner novo deve ser criado e inicializado com o código da função e todas as dependências necessárias antes que a execução da função possa começar; com funções de testes vazias, para que medisse o tempo necessário de configuração do ambiente de execução da função e todos os testes foram executados em lotes.

Para que os testes fossem realizados foi criada uma estrutura que permitiu a coleta consistente e automatizada de métricas para a realização da pesquisa, chamada *Serverless Performance Framework* (SPF). Os detalhes da arquitetura do SPF estão descritos na Figura 3. Nos componentes da arquitetura do SPF, todas as funcionalidades para gravação, cálculo e análise de métricas de desempenho e custo são expostas por meio de uma API padrão, nos componentes são utilizadas uma função *Lambda* do *.NET Core 2* "*Metrics Lambda"* que armazena as métricas; uma função *NodeJS "Cost Lambda"* que calcula o custo estimado de uma função e utiliza tabelas do *DynamoDB* para armazenamento. Nos componentes da *AWS* são realizados testes específicos nas funções *Lambda* vazias de cada linguagem e depois estas

se conectam com os componentes comuns através da *API* com uma função *AWS Lambda* do *NodeJS.* Os componentes do *Azure,* assim como na *AWS*, também armazenam métricas por meio da mesma *API*, e as funções de testes foram configuradas para integrar-se ao *Azure Applications Insights*, que coleta dados avançados de *log* e telemetria.

Figura 3 - Arquitetura do SPF.



Fonte: (CLYNCH; JACKSON; 2018).

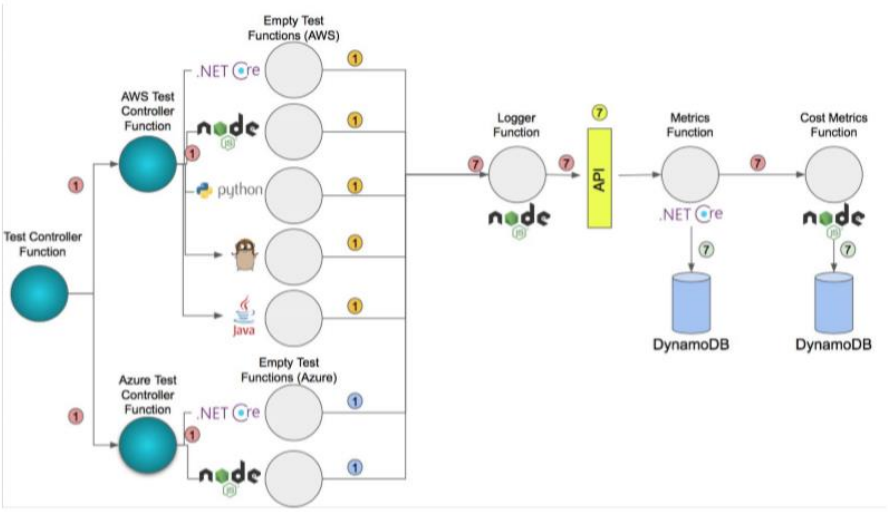
Os intervalos de *Warm-Start* da *AWS* tiveram quatro testes separados de inicialização a execução de 1 hora de duração, no total foram 248 testes individuais executados contra cada um dos cinco tempos de execução das linguagens. Os resultados mostraram que o *Python* teve a melhor média de tempo de execução. Já os testes de *Cold-Start* mostraram resultados um pouco diferentes em relação ao *Warm-Start*, duraram 144 horas, envolvendo 144 invocações individuais das funções de testes vazias de cada tempo de execução. O *Python* continuou com o menor tempo de execução seguido do *Go.*

Os testes de *Warm-Start* do *Azure* tiveram um total de 273 testes com 4,5 horas para os dois tempos de execução. Os resultados mostram que o *C#* executa consistentemente mais rápido. Já nos testes de *Cold-Start* que duraram 6 dias, o *C#* teve um desempenho melhor que o *NodeJS* na Azure.

Partindo de seus resultados os autores reconhecem que a melhor escolha no *AWS Lambda* é o *Python* por ter apresentado bons resultados independentes do tipo de teste. Da mesma forma, o *C# .NET* é a melhor opção no *Azure Functions.* Além disso a pesquisa gerou custos baseados em um ecossistema desenvolvido no modelo *CostHat* para o SPF, no qual mostrou que o rendimento combinado geral de 30k TPS é realista e é possível uma alta taxa de

cenários de inicialização a frio. Portanto, o trabalho nos mostra o quão se pode refinar e adequar composições de funções em aplicativos sem servidor, fazendo com que a solução se torne mais flexível e econômica atendendo a cada especificidade.

Figura 4 - Modelo CostHat do SPF modificado.



Fonte: (CLYNCH, 2018)

2.5 Avaliação

A partir da análise dos trabalhos anteriores conseguimos avaliar quais serviços da AWS estão sendo utilizados, assim como, os componentes responsáveis pela estruturação de cada algoritmo de álgebra linear implementado. Essa análise nos faz identificar as linguagens utilizadas, o conjunto de serviços *serverless* e não *serverless* e como cada trabalho lida com a orquestração das funções. Dessa forma, foi desenvolvida a tabela abaixo que facilita a avaliação de todas essas questões analisadas por nosso trabalho. O último trabalho discutido não consta na tabela, pois não se trata de uma aplicação científica, mas apenas a avaliação da execução de funções vazias. Entretanto, sua discussão foi importante pois revela que a escolha da linguagem influencia no desempenho da aplicação em Computação Sem Servidor.

Tabela 1 - Comparação dos trabalhos apresentados.

Solução	Serviços AWS	Componentes não *Serverless*	Algoritmos Implementados	Linguagem Utilizada	Orquestração das Funções
Trabalho 01	AWS *S3;* AWS *Lambda;*	-	*Strassen*	*Python*	AWS *Step Functions*
Trabalho 02	*Amazon SQS; AWS Lambda; AWS S3;*	*LAmbdaPACK; Redis;*	Cholesky; GEMM; QR; SVD;	*Python*	*LAmbdaPACK*

Trabalho 03	*AWS Lambda; AWS S3;*	-	Previsão de precipitação	*Python*	*Snafu*

Fonte: elaboração própria.

Como pode ser visto na tabela, os trabalhos se igualam em algumas ferramentas e algoritmos utilizados. Por exemplo, temos o *AWS S3,* que é utilizado como armazenamento e *Python* que é escolhida como linguagem comum entre os três trabalhos, nos quais realizam cálculo numéricos como experimento. Isso nos faz concluir que os serviços destacados na tabela são aptos a serem utilizados em nosso trabalho, assim como vemos espaço para analisar outras linguagens na execução de algoritmos de álgebra linear. Em relação aos algoritmos implementados, a Multiplicação de Matrizes é a base de dois dos trabalhos, então é uma aplicação representativa dos desafios de computação de alto desempenho. Qual linguagem seria a mais indicada para Computação de Alto Desempenho no paradigma *serverless* é uma questão em aberto.

3 FUNDAMENTAÇÃO TEÓRICA

Nesta seção apresentamos os referenciais teórico do trabalho apresentando e discutindo o contexto de computação de alto desempenho e seus requisitos, para em seguida apresentar um pouco sobre computação de alto desempenho, computação em nuvem e o paradigma sem servidor.

3.1 Computação de Alto de Desempenho

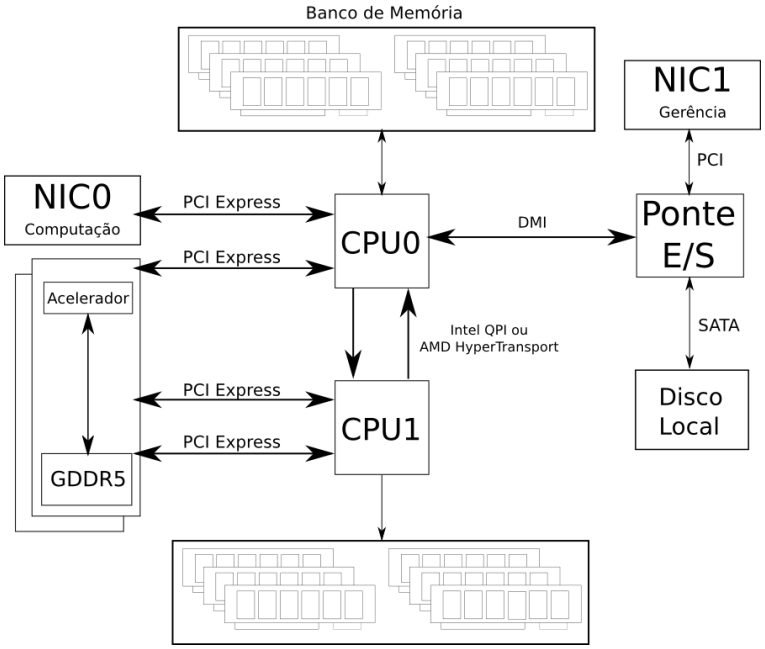
A definição da indústria para Computação de Alto Desempenho é que geralmente se refere a prática de agregar poder computacional de forma que seja entregue um desempenho muito maior do que o disponível em uma estação de trabalho típica com o objetivo de resolver grandes problemas na ciência, engenharia ou negócios[1]. Portanto, trata-se de um sistema distribuído, formado por vários nós de processamento dada a agregação de poder. Entretanto, a distribuição não é geográfica: os *clusters* para HPC são abrigados em *datacenters* e sua interconexão são redes de alta velocidade para diminuir a latência na comunicação de processos.

Na Figura 5, temos a representação de um elemento de processamento de um supercomputador moderno. Vários desses elementos são interligados para formar um cluster HPC. Vemos que existem mais de um processador, cada um com vários núcleos. Há também a

[1]https://insidehpc.com/hpc-basic-training/what-is-hpc/

presença de placas aceleradoras como GPUs ou FPGA, além do acesso otimizado à rede de comunicação. O objetivo de apresentar essa figura é mostrar como os detalhes do *hardware* são complexos, sendo que para utilizar os modelos de programação para computação paralela em um supercomputador moderno a fim de extrair o máximo de desempenho, o desenvolvedor precisar estar ciente dos detalhes. Uma solução capaz de abstrair tais informações do processo de desenvolvimento seria beneficia para a produtividade dos pesquisadores que fazem uso da HPC para resolver problemas de larga escala.

Figura 5 - Arquitetura interna de um nó de processamento utilizado para computação de alto desempenho.



Fonte: (DONGARRA, 2013)

3.1.1 Modelos de Programação

Para desenvolver aplicações de HPC, em geral, há dois modelos clássicos: memória compartilhada e troca de mensagens. No modelo de variáveis compartilhadas, tarefas concorrentes compartilham um espaço de endereçamento comum, no qual elas podem escrever e ler de maneira assíncrona. Essas tarefas podem ser representadas por processos ou *threads*. Sendo as tarefas processos, como no caso de arquiteturas *Non Uniform Memory Access* (NUMA), uma biblioteca pode ser utilizada para mapear regiões compartilhadas da memória entre processos diferentes. Por exemplo, na biblioteca POSIX[2], esse mapeamento resulta em um arquivo virtual que representa a região compartilhada. O sistema operacional deve oferecer mecanismos de travas ou semáforos para o controle de concorrência.

[2] https://pubs.opengroup.org/onlinepubs/9699919799/

No paradigma de troca de mensagens, a execução de uma aplicação científica é intercalada entre períodos de computação e comunicação. Durante a computação, várias tarefas ou processos executam acessando apenas a memória local da máquina na qual residem. Após a fatia local dos cálculos ser concluída, na fase de comunicação, tarefas em máquinas diferentes trocam mensagens para sincronização e nova divisão do trabalho restante. A comunicação é feita através de chamadas a funções de uma biblioteca. Em geral, a transferência de dados requer cooperação entre tarefas para ocorrer. Por exemplo, uma operação para enviar dados, invocada por um processo, precisa da invocação de uma operação correspondente em outro processo para recebê-los. A comunicação também pode ocorrer envolvendo várias tarefas invocando uma operação coletiva de troca de dados.

Em ambos os modelos, o desenvolvedor precisa considerar detalhes da arquitetura subjacente. No modelo de memória compartilhada, é preciso saber se os nós de processamento do *cluster* são *multicores* e ter noção da quantidade de núcleos, além da hierarquia da memória. Já na troca de mensagens, a qualidade da rede de interconexão é importante para definir a granularidade dos processos e o impacto da comunicação no tempo total de execução.

3.1.2 Granularidade de Aplicações

Apesar dos dois modelos apresentados serem a base para a maioria das análises de computação de alto desempenho, considerando o aspecto das aplicações em si e o nível de abstração que os desenvolvedores desejam atuar, podemos considerar uma aplicação HPC como um conjunto de tarefas. Uma tarefa seria uma sequência de instruções que atuam em conjunto como um grupo (MATTSON G., SANDERS A. e MASSINGILL L., 2004). O desenvolvedor decompõe suas aplicações em tarefas, que por sua vez são mapeadas em recursos chamados elementos de processamento (núcleos de CPU, nós de processamento etc.) de acordo com sua granularidade (quantidade de trabalho).

As tarefas podem se comunicar entre si entre dois extremos, de forma regular e intensa (aplicações fortemente acopladas) ou apenas no início/fim da computação (*embarrassingly parallel*). A maioria das aplicações se encaixa em um ponto intermediário desse espectro, apresentando aspectos de acoplamento ou desacoplamento.

Aplicações de álgebra linear computacional estão entre as que mais se beneficiam da computação de alto desempenho (ANDREWS, 2000), inclusive, são a base para *benchmarks* utilizados para pedir o poder dos maiores supercomputadores da atualidade (A, C., *et al.*, 2018). Nem todas as aplicações desta área podem ser consideradas desacopladas, entretanto a multiplicação de matrizes pode ser modelada de forma que as tarefas não precisem se comunicar

durante o cálculo. Por exemplo, considere a multiplicação C = A x B, matrizes de ordem n. Podemos decompor esse cálculo na seguinte definição de tarefas:

Código 1 - Decomposição da Multiplicação de Matrizes

```
tarefa [i  = 0 até n -1] {
   double a[n];      # linha i  da matriz A
   double b[n , n]; # toda a matriz B
   double c[n];      # linha i da matriz C
   for [j = 0 até n – 1] {
      c[j] = 0.0;
      for [k = 0 até n – 1] {
         c[j] = c[j] + a[k] * b[k , j];
      }
   }
}
```

Fonte: elaboração própria.

Neste exemplo, cada linha de C pode ser calculada por uma tarefa independente. A única restrição é que as matrizes devem ser informadas para todas as tarefas ao início da computação, com algum particionamento envolvido, e depois cada tarefa deve contribuir seu cálculo para a matriz final. Não se trata sequer da comunicação entre tarefas, mas sim entre uma entidade coordenadora e as tarefas.

Nos *clusters* tradicionais, as tarefas da multiplicação de matrizes podem ser alocadas para os elementos de processamento tradicionais como núcleos, nós de processamento e aceleradores. Entretanto, como explicaremos nas seções seguintes, as nuvens computacionais fornecem novas abstrações de processamento que podem ser utilizadas na computação de alto desempenho.

3.2 Computação em Nuvem

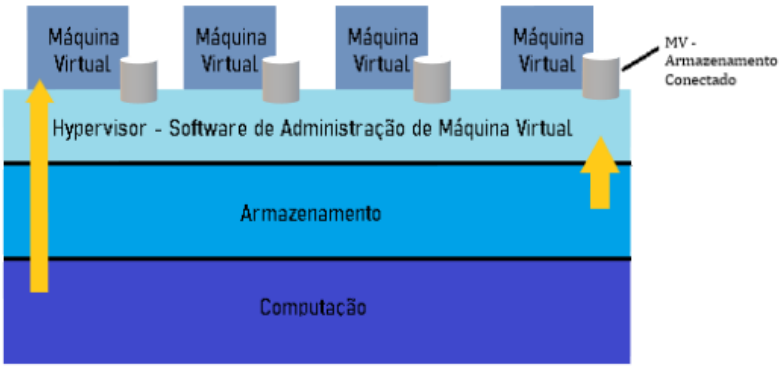
Uma das definições mais celebradas de Computação em Nuvem é dada por (MELL e GRANCE, 2011), na qual Computação em Nuvem é um modelo para permitir acesso onipresente, conveniente e de rede sob demanda de um conjunto de serviços configuráveis através da Internet.

3.2.1 Vantagens e Motivação

Em sintonia com a definição apresentada, de acordo com (PIPER e CLINTON, 2019), computação em nuvem nos traz diversos benefícios. Alguns destes são os inúmeros recursos altamente disponíveis e escaláveis que nos permite projetar várias camadas de redundância, nas quais cargas de trabalho podem ser remanejadas de forma automática, com componentes que falharam sendo substituídos na íntegra instantaneamente. Além disso, a nuvem possui milhares de servidores espalhados em regiões diferentes pelo mundo, permitindo assim que possamos conectar recursos em regiões diferentes, o que é importante quando temos falha de uma região completa, desta forma o cliente não tem perda total de seus serviços. A nuvem oferece acesso a tudo isso sob demanda por um preço muito inferior ao que custaria em um ambiente local comparável.

Com uma rede de servidores tão grande das plataformas da nuvem, a segurança é um ponto fundamental a ser tratado. A nuvem oferece infraestruturas protegidas profissionalmente, e assume a "responsabilidade pela segurança da rede subjacente e da infraestrutura de computação [...] " (PIPER e CLINTON, 2019). O que nos permite de se utilizar dos recursos computacionais sob demanda na nuvem é a virtualização, como apresentada na Figura 6. Por exemplo, partindo das preferências de máquina virtual do cliente, a infraestrutura irá extrair recursos necessários de dispositivos existentes maiores para atender o determinado cliente. Esse modelo de virtualização nos oferece velocidade, utilizar os servidores virtuais da nuvem nos poupa tempo em relação a utilização de servidores físicos nos quais precisam ser definidos, adquiridos, provisionados, testados e lançados, processos estes que podem levar meses para serem concluídos em servidores locais. Outro benefício da virtualização é a eficiência, os servidores físicos da infraestrutura da nuvem estão sempre com diversas máquinas virtuais alocadas em si, isto faz com que esses sempre sejam preenchidos com cargas de trabalho virtuais, quando um servidor atinge seu limite de capacidade as cargas de trabalho de *overflow* podem ser movidas para uma outra máquina.

Figura 6 - Máquinas virtuais acessando recursos de armazenamento e computação de seu servidor hospedeiro.



Fonte: adaptado de (PIPER e CLINTON, 2019).

A nuvem é escalonável, possuir este quesito nos faz possuir capacidade para sermos atendidos perfeitamente a quaisquer mudanças ocorridas na demanda, ou seja, sempre que a demanda exigir mais poder de computação temos como resposta a inclusão de novas instâncias de servidor para atendê-la. A nuvem também é elástica, este quesito nos proporciona a capacidade de redimensionamento de forma fácil e automática, no qual, ao não necessitar mais desse redimensionamento causado pelo escalonamento, o serviço retorna as suas definições padrões, geralmente o cliente define um limite mínimo e máximo para suas aplicações.

3.2.2 Modelos de Implantação

A computação em nuvem é dividida entre dois modelos de implantação principais, nuvem pública e nuvem privada, sendo que na nuvem pública o provedor possui infraestrutura compartilhada entre diversos clientes; e na nuvem privada a infraestrutura entregue é de uso exclusivo de apenas um cliente. Exemplos: nuvem privada (*OpenStack*) (ROSADO e BERNARDINO, 2014); e nuvem pública (*AWS*[3], *Azure Microsoft* [4] e *Google Cloud*[5]). Na nuvem privada, a instituição permanece responsável pela manutenção da infraestrutura física, incluindo as questões de segurança e fornecimento de energia e refrigeração. Entretanto, as interfaces do sistema de nuvem são mais flexíveis, permitindo aos desenvolvedores e operadores um controle dinâmico do parque computacional.

Uma terceira possibilidade são infraestruturas híbridas, nas quais instituições combina seus recursos locais com recursos de nuvens públicas. As razões são diversas, dentre elas: auxiliar um processo de migração total para a nuvem; manter uma parte das aplicações em

[3] https://aws.amazon.com/pt/
[4] http://azure.microsoft.com/pt-br/
[5] https://cloud.google.com/

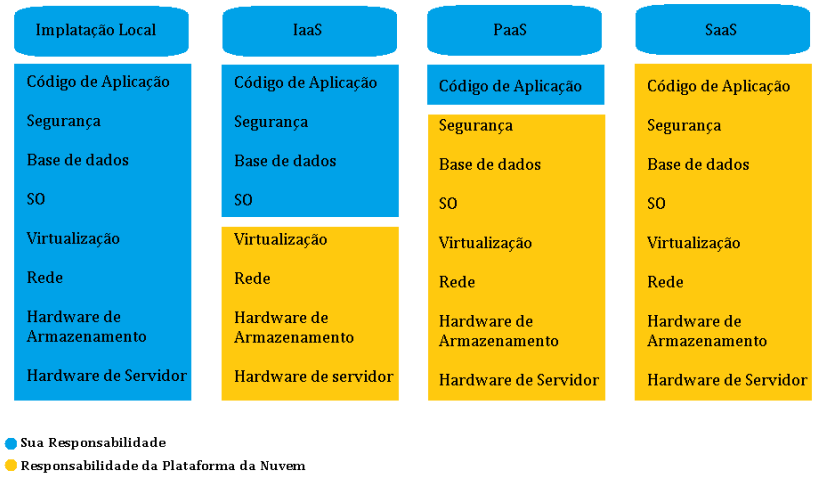
execução local para atender requisitos regulatórios; oferecer rapidez e latência menor para usuários locais de acordo com a necessidade de aplicações sensíveis a esses critérios. A maioria dos serviços de nuvem pública oferecem soluções para a criação de nuvens híbridas.

3.2.3 Modelos de Serviços

A computação em nuvem apresenta três modelos de serviços: IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*) e SaaS (*Software as a Service*). Na Infraestrutura como Serviço (IaaS) o cliente contrata uma capacidade de *hardware* para utilizá-lo como quiser; em Plataforma como Serviço (PaaS) o cliente pode contratar uma plataforma para criar, hospedar e gerir seu aplicativo de acordo com uma pilha de *software* pré-definida; e em *Software* como Serviço (SaaS) o cliente pode se utilizar de um *software* sem comprar sua licença e sem considerar os requisitos de infraestrutura.

Nos três modelos o cliente só paga pelo que se utilizar de acordo com um contrato disponibilizado pelo provedor da solução, em geral definido de acordo com responsabilidade compartilhada (ver Figura 7). Dependendo do modelo abordado, cliente e provedor sabem quais são suas obrigações e direitos, incluindo taxas de cobrança e nível de suporte oferecido. É importante observar que mesmo no modelo IaaS, o provedor já assume várias responsabilidades, o que facilita a utilização da infraestrutura pelo cliente.

Figura 7 - A divisão da responsabilidade em vários tipos de infraestrutura.



Fonte: adaptado de (PIPER e CLINTON, 2019)

O paradigma *Serverless*, tópico deste trabalho, é uma abordagem no paradigma PaaS, porém com um foco ainda maior na abstração de recursos. Cada provedor de nuvem citado anteriormente oferece serviços em cada modelo, na *AWS*: IaaS (EC2), PaaS (*Elastic*

BeanStalk) e SaaS (*Amazon Chime*); no *Google Cloud*: IaaS (*Google Compute Engine*), PaaS (*Google App Engine*) e SaaS (*Google* Drive); e no *Azure*: IaaS (*Azure Virtual Machines*), PaaS (*Microsoft Azure* Cloud Services) e SaaS (*Microsoft Office 365*). Também temos as opções de funções sem servidor *AWS Lambda*, *Google Cloud Functions* e *Azure Functions*.

3.3 Computação Sem Servidor

Nesta seção apresentamos os fundamentos da Computação Sem Servidor (seção 3.2.1), seus benefícios (seção 3.2.2), desafios (3.2.3) e apresentamos um exemplo de arquitetura de aplicação *serverless* (Subseção 3.2.4). A visão apresenta é geral, apresentando o contexto comum de utilização de computação sem servidor, para depois abordar como a HPC pode se beneficiar desse novo paradigma.

3.3.1 Fundamentos de Computação Sem Servidor

Serverless Computing, ou traduzido Computação sem Servidor, é uma abordagem na qual os clientes não precisam administrar os servidores nos quais seus aplicativos são executados. Esse paradigma traz o aumento da abstração aos desenvolvedores, em relação ao *hardware* da infraestrutura, o que facilita no uso da nuvem para eles.

O início da virtualização na nuvem veio com as duas empresas pioneiras do ramo, Amazon e Google, com os serviços respectivos – Amazon EC2 e *Google App Engine* – que traziam abordagens distintas. Com o Amazon EC2 o cliente trabalhava com instâncias nas quais se aproximavam visualmente de seu computador local, ou seja, o cliente era o próprio responsável pelos servidores. Já o *Google App Engine* possuía uma abordagem específica de domínios de aplicativos, com uma interface mais amigável sem a necessidade de gerenciamento de servidores, na qual já possuía características de uma PaaS. Dentre as duas abordagens o Amazon EC2 encontrou mais aceitação dentre os desenvolvedores inicialmente, visto que apresentava um caminho mais natural de migração sem reescrita de código das aplicações.

Após a comunidade reconhecer os requisitos necessários para se utilizar dessas abordagens, a Amazon lançou, em 2015, com uma nova proposta, a AWS *Lambda*. Esse serviço oferecia funções na nuvem, nas quais os desenvolvedores não precisariam mais se preocupar com os servidores, adotando o conceito de Computação sem Servidor. De acordo com (JONAS, SMITH, *et al.*, 2019), um serviço é considerado sem servidor se for dimensionado automaticamente, sem necessitar de provisionamento explícito e for cobrado com base em seu

uso. Além disso, o autor indica que a junção de *back-end* como Serviço (BaaS) e Função como Serviço (FaaS) é a computação sem servidor.

3.3.2 Benefícios da Computação Sem Servidor

Os benefícios da migração da computação com servidor para a computação sem servidor se dão como na transição de um código feito em linguagem de baixo nível para uma linguagem de alto nível, ou seja, em visão de abstração. A computação sem servidor torna-se significativamente inovadora em relação a plataforma como serviço (PaaS), pois se "[...] difere de seus antecessores de várias maneiras essenciais: melhor escalonamento automático, isolamento forte, flexibilidade de plataforma e suporte ao ecossistema de serviços." (JONAS, SMITH, *et al.*, 2019).

A Computação sem Servidor atrai os desenvolvedores por suas inúmeras facilidades. Com ela não é necessário entender a infraestrutura da nuvem, os desenvolvedores passam a focar mais em seus códigos de função, além de seu custo no qual o cliente paga apenas pelo que usar, podendo levar à economia de recursos.

3.3.3 Desafios para Computação Sem Servidor

Apesar dos benefícios, há questões a serem tratadas na criação de aplicações em Computação sem Servidor. Por exemplo, em Jonas et al. (2019) realizou experimentos que se deu a partir da análise das execuções de aplicativos sem servidor de alto desempenho, de diferentes domínios. As cinco aplicações estudadas necessitam de dimensionamento automático de baixa granularidade, um forte motivo para que se utilizassem de Computação sem Servidor.

Dos resultados dos experimentos, o autor indica em seu trabalho quatro obstáculos que impede as ofertas de computação sem servidor de obterem seus desempenhos em níveis máximos. O primeiro indicativo é o armazenamento inadequado para operações refinadas, o autor indica que isso ocorre principalmente pelas limitações impostas pelos serviços de armazenamento oferecidos pelos provedores. Geralmente serviços de armazenamento de custo barato possuem alto custo de acesso e alta latência de acesso. Por esse e outros motivos o autor motiva o desenvolvimento de opções mais eficientes e duráveis de armazenamento.

Outro desafio identificado é a falta de coordenação refinada, que acontecia pela falta de suporte a aplicativos com estado, mas desde o trabalho citado anteriormente, surgiram alguns avanços relacionados à coordenação refinada, por exemplo, o serviço AWS *Step Functions* que permite a criação de máquinas de estado para coreografia de invocações de

funções. O terceiro obstáculo vem como o baixo desempenho para padrões de comunicação que ocorrem por exemplo entre os padrões de *broadcast*, agregação e *shuffle*, muito utilizados por aplicativos. O último obstáculo foi o desempenho imprevisível que acaba mudando para cada aplicativo.

Após a avaliação do comportamento das aplicações, o autor listou cinco desafios nas áreas de abstrações, sistemas, rede, segurança e arquitetura. Na área de abstração os desafios são nos requisitos de recursos, no qual a abstração utilizada não permite que a plataforma saiba requisitos específicos de determinados aplicativos; além da abstração na dependência dos dados entre as funções, isso faz com que por exemplo torne padrões de comunicações ineficientes.

Em sistemas de suporte, é indicado o desafio de armazenamento de alto desempenho, configurável e provisionado de forma transparente. As necessidades surgiram entre o armazenamento efêmero sem servidor e o durável sem servidor, que são extremamente necessários para a coordenação das funções e armazenamento de dados necessários para a execução, respectivamente. O desafio de latência é voltado para a sobrecarga significativa imposta pela comunicação entre as funções. Em segurança os desafios estão entre o planejamento da randomização e o isolamento físico, contextos de segurança refinados e a computação sem servidor inconsciente.

Os desafios de arquiteturas de computadores são voltados para a heterogeneidade, preço e facilidade de gerenciamento de *hardware*, a preocupação surge com o desempenho dessas plataformas que não está evoluindo com a mesma velocidade do passado e provavelmente não tenham um aprimoramento significativo nos próximos anos. (JONAS, SMITH, *et al.*, 2019) acredita que esses desafios são fator chave para trazer mais vantagens aos serviços sem servidor.

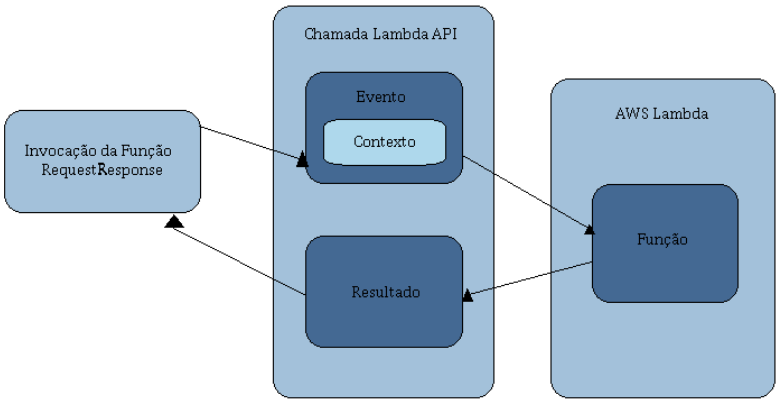
3.3.4 AWS Lambda

Com o objetivo de conceitualizar de forma mais concreta os conceitos de computação sem servidor, discutimos a seguir uma das suas implementações mais utilizadas. De acordo com Poccia (2017), o AWS *Lambda*, serviço no qual iremos utilizar para executar nossas funções em *Go* e *Python*, é diferente das abordagens tradicionais existentes que são baseadas em servidores físicos ou virtuais, pois desenvolvedores se preocupam apenas em fornecer a lógica da computação, agrupada em funções, e o próprio serviço se encarrega de executar as funções sempre que necessário, fazendo a parte de gerenciamento da pilha de *software* usada pelo ambiente de execução escolhido. No presente trabalho os ambientes de execução são de *Go* e *Python*, possuindo a disponibilidade e escalabilidade existente na nuvem.

Com o *Lambda* temos que cada função possuirá sua própria configuração que irá se utilizar dos recursos de segurança padrão da plataforma AWS definindo o que cada uma pode fazer e em quais recursos. O AWS *Lambda* é um serviço *serverless*.

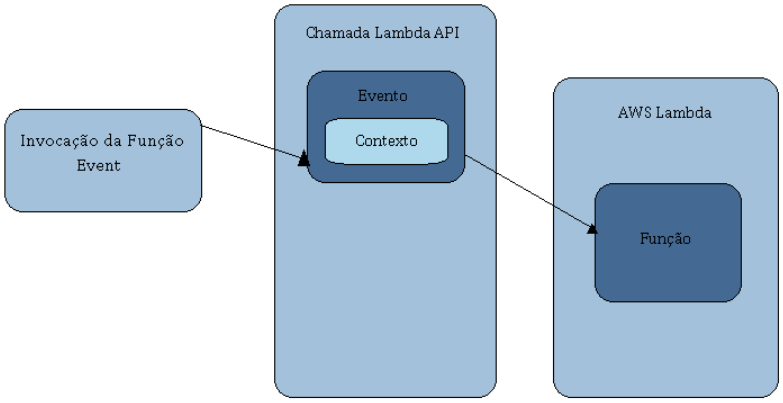
O *Lambda* executa as funções em contêineres que " [...] são um método de virtualização de servidor em que o *kernel* do sistema operacional implementa vários ambientes isolados." (POCCIA, 2017). As funções são orientadas a eventos, podemos executá-las diretamente pela própria plataforma AWS, ou podemos inscrevê-las em eventos gerados por outros recursos ou serviços. Por exemplo, podemos inscrever uma determinada função em um recurso de repositório de arquivos como o S3, ao adicionarmos um arquivo nesse repositório nossa função será executada automaticamente. Da mesma forma podemos escolher diversos outros recursos para disparar nossa função, cabe ao desenvolvedor direcionar quais recursos para cada situação. As funções também podem ser invocadas de maneira síncrona e assíncrona. A maneira síncrona (ver Figura 8) ocorre através do tipo de invocação *RequestResponse*, recebendo uma entrada e um contexto como evento e retornando um resultado. Já de maneira assíncrona (ver Figura 9), a chamada retorna imediatamente e nenhum resultado é retornado, enquanto a função continua seu trabalho, o tipo de invocação é *Event*.

Figura 8 - Chamando uma função AWS Lambda de forma síncrona com o tipo de invocação RequestResponse.



Fonte: adaptado de (POCCIA, 2017)

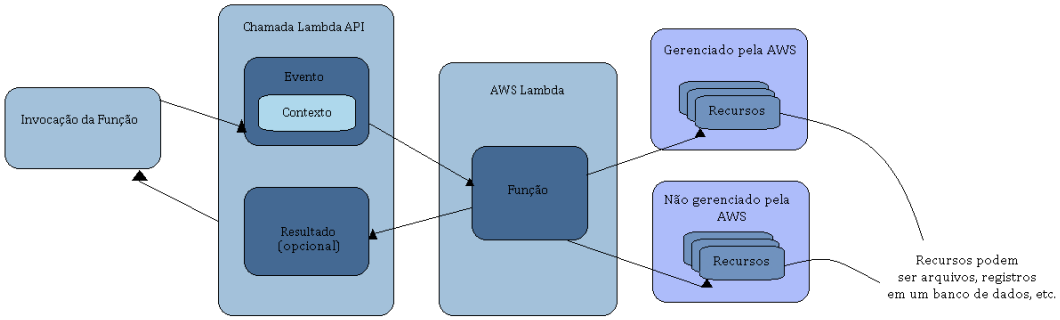
Figura 9 - Chamando uma função AWS Lambda de forma assíncrona com o tipo Event.



Fonte: adaptado de (POCCIA, 2017)

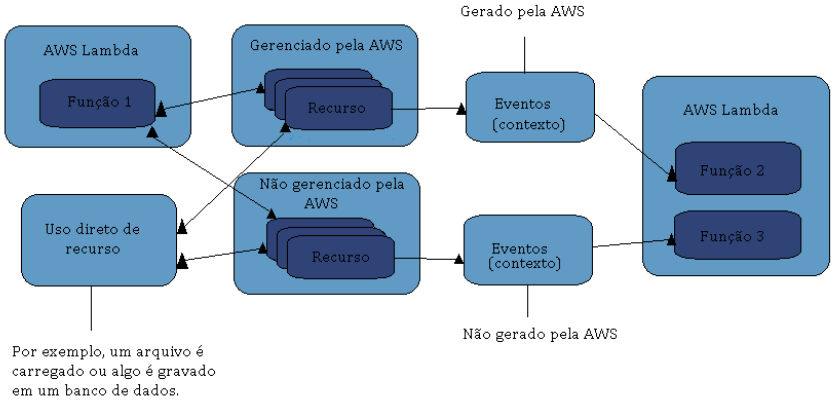
Invocações assíncronas são úteis principalmente quando são utilizadas para acessar e modificar o status de algum recurso ou chamar outro serviço, como mostra a Figura 10. Quando as funções são inscritas em eventos gerados por outros recursos ocorre uma chamada assíncrona quando os eventos escolhidos são gerados, passando estes como entrada para a função (Figura 11).

Figura 10 - As funções podem criar, atualizar ou excluir outros recursos. Os recursos também podem ser outros serviços que podem realizar algumas ações, como enviar um e-mail.



Fonte: adaptado de (POCCIA, 2017)

Figura 11 - As funções podem se inscrever em eventos gerados pelo uso direto de recursos ou por outras funções interagindo com recursos. Para recursos não gerenciados pela AWS, você deve encontrar a melhor maneira de gerar eventos para inscrever funções para esses recursos.



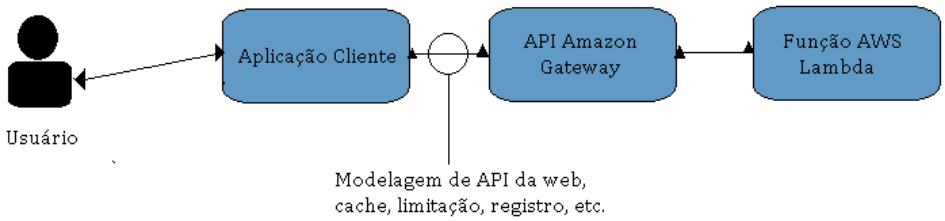
Fonte: adaptado de (POCCIA, 2017)

Nos exemplos anteriores, não é abordado diretamente como um aplicativo cliente interage com as funções *Lambda*, pois foi presumido um tipo de invocação direta. Mas, existe uma API específica para fazer esses acessos, através da operação *Invoke*. Para que possa ser chamada, é necessário que o aplicativo cliente tenha as permissões corretas para invocar determinada função, pois são aplicadas verificações de segurança, e através de credenciais da AWS para a autenticação é verificado se as invocações possuem autorização para executar essa *Invoke* em uma determinada função ou recurso. Claro que não é recomendado adicionar credenciais diretamente em aplicativos clientes, ou seja, em algo que irá ser oferecido ao usuário final, pois um usuário avançado pode encontrar essas credenciais e comprometer sua aplicação, por conta disso é importante possuir um serviço de gerenciamento de autenticação, como *Amazon Cognito*, que irá facilitar todo acesso a sua aplicação.

É possível substituir o acesso direto dos clientes através da operação AWS *Lambda Invoke* por uma própria API da Web que pode ser construída mapeando o acesso às funções *Lambda* para URLs e métodos HTTP mais genéricos. Podemos utilizar o serviço Amazon *API Gateway* que mapeia o acesso a um recurso específico, através de uma URL com um método HTTP para invocação de uma função. Não é preciso possuir funções diferentes para cada recurso e combinação de método HTTP, pois o recurso e o método são enviados como parte dos parâmetros de entrada de uma única função que processa e entende o que foi disparado. Além disso, o API *Gateway* inclui resultados de *cache* que produz redução de carga no *back-end*, controle de fluxo para evitar sobrecarga em picos, gerenciamento de chaves e outros recursos. Utilizar essa API (*Gateway*) traz o desacoplamento do uso direto do cliente às funções, "[...] expondo uma API *web* limpa que não pode ser consumida por serviços externos que não

deveriam ter conhecimento da AWS" (POCCIA, 2017). Com o API *Gateway* podemos também oferecer acesso público a APIs, ou seja, nenhuma credencial é necessária para seu acesso, ou seja, podemos criar sites públicos cujos URLs são dinamicamente servidos por funções *Lambdas* (Figura 12). Utilizar o Amazon Cognito em conjunto com o Amazon API Gateway para gerenciar a autenticação e autorização para os clientes é uma alternativa.

Figura 12 - Usando o Amazon API Gateway para dar acesso público a uma API e criar sites públicos apoiados pela AWS Lambda.



Fonte: adaptado de (POCCIA, 2017)

Outro ponto importante é o modelo de custo do AWS *Lambda* que é gerado através do número de invocações de sua função e pelo tempo de execução de todas as invocações, sendo que irá depender também da memória alocada para essa função já que os custos de tempo de execução crescem linearmente com a memória. Ao desenvolvedor, resta decidir a quantidade máxima de memória a ser utilizada, sendo que quanto mais memória alocada, automaticamente a AWS atribui mais recursos de CPU. Por exemplo, uma função lambda que executa com limite de 512 MB de RAM tem menos *slots* de alocação de CPU do que uma função lambda com 3GB de RAM. No momento da escrita deste trabalho, o limite de memória para uma única invocação de função é 10 GB de RAM[6]. O resultado direto é que invocações de funções com mais memória tem mais custo do que invocações com menos memória.

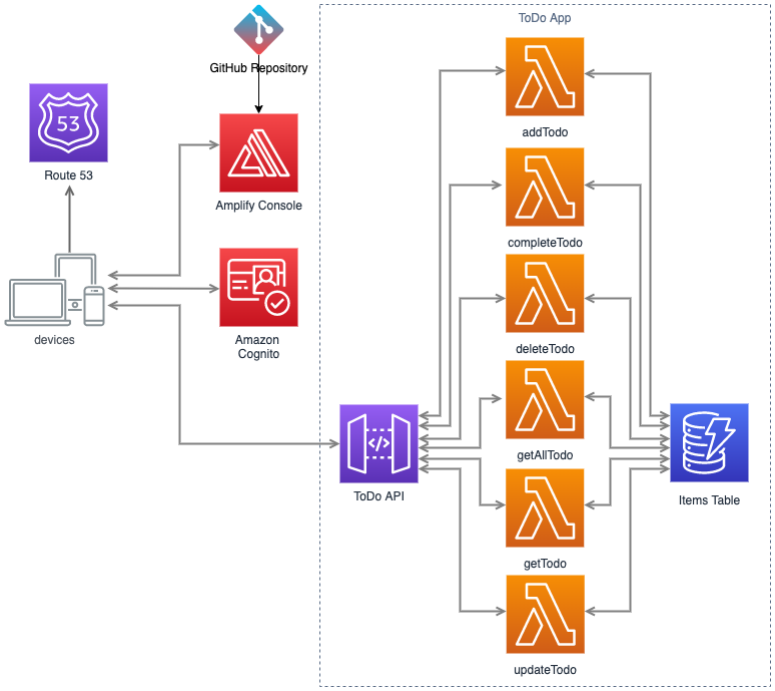
3.3.5 Exemplo de Arquitetura de Aplicação Computação Sem Servidor

Já vimos as vantagens e os desafios para adoção de arquiteturas *serverless*, mas para entendermos melhor a arquitetura, podemos analisar o diagrama de uma aplicação simples que mantêm listas de tarefas (MAGALHÃES, 2020), permitindo ao usuário criar, atualizar, visualizar os itens existentes e excluir itens, se necessário. A arquitetura geral da aplicação está na Figura 13. Essa aplicação *web* orientada a eventos, se utiliza da AWS *Lambda e* da *Amazon*

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

API Gateway para sua lógica de negócios. Ela também usa o *Amazon DynamoDB* como banco de dados e o *Amazon Cognito* para gerenciamento de usuários. Todo o conteúdo estático é hospedado usando o AWS *Amplify Console* e o *Amazon Route 53* é um serviço de *DNS*. Esta não é uma aplicação HPC, tema do nosso trabalho, entretanto seu estudo é importante para compreender as decisões de projeto da AWS *Lambda*.

Figura 13 - Diagrama da Arquitetura de uma aplicação web de listas de tarefas.



Fonte: Repositório GitHub. [7]

Após estar registrado e autenticado, um usuário pode requisitar serviços que necessitam de comunicação com o *back-end*, com isso, a aplicação emite chamadas da *API REST* (*ToDo API*) para o *back-end*. O *back-end* é onde a lógica de negócio realmente é implementada através de funções *lambdas* que são invocadas por eventos emitidos pela *API REST* do *API Gateway*. Nessa aplicação temos seis funções *lambdas* responsáveis por aspectos diferentes, as quais salvam os itens necessários em tabelas do DynamoDB (*Items Table*). As funções não estão em execução em uma máquina virtual ou contêiner específico, é a criação dos eventos que inicia a instanciação do código da função em um ambiente da nuvem, sem que o usuário ou desenvolvedor tenha que configurá-lo.

O contexto da aplicação *web* descrito é diferente do cenário da computação de alto desempenho descrito na seção 3.1. Um dos objetivos desse trabalho é conciliar esses cenários.

[7] https://github.com/aws-samples/lambda-refarch-webapp

A nossa proposta, já abordada pelos trabalhos relacionais na seção 2, é utilizar funções lambdas para representar tarefas de uma aplicação de computação de alto desempenho fracamente acoplada. Enquanto os trabalhos relacionados discutem a visão arquitetural de mais alto nível, como o exemplo da arquitetura na Figura 13, buscamos responder qual ambiente de execução, *Python* ou *Go*, é o mais adequado para a codificação a nível de tarefa. Por exemplo, na multiplicação de matrizes, buscamos responder qual linguagem fornece melhor desempenho para computar um elemento ou linha da matriz resultado.

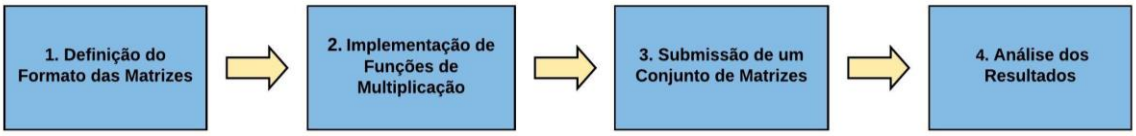
4 PROCEDIMENTOS METODOLÓGICOS

Como apresentado na seção Objetivos, nosso objetivo principal é realizar uma comparação entre as linguagens de programação *Python* e *Go,* e isso será realizado por meio do algoritmo de Multiplicação de Matrizes no serviço de computação sem servidor *AWS Lambda*. Para realizar esta análise desenvolvemos as etapas descritas a seguir.

4.1 Etapas de experimento

As etapas da metodologia são apresentadas nas subseções (4.1.1 a 4.1.3) seguintes na ordem de execução na Figura 14. A exceção é a última etapa, apresentada no próximo Capítulo.

Figura 14 - Fluxograma das etapas metodológicas



Fonte: autoria própria.

4.1.1 Definição do Formato das Matrizes

Desenvolvemos o experimento por meio do algoritmo de Multiplicação de Matrizes tradicional, nas linguagens *Go* e *Python*. Nessa primeira etapa já consideramos os objetivos específicos 1 e 2 completados, pois através da análise dos trabalhos da seção 2 é possível concluir a viabilidade desses objetivos.

Um pré-requisito para nossos experimentos condiz em definir o formato da matriz utilizado nas funções. Nossa opção foi utilizar matrizes densas, preenchidas com valores de ponto flutuante, usando o tipo correspondente em *Go* e usando a notação adequada em *Python*. As matrizes submetidas para multiplicação foram matrizes quadradas, preenchidas com o valor 1.0 em todos os elementos. Essa opção não tem impacto no desempenho, visto que a

representação interna do número é a mesma seja qual for o valor armazenado, além de permitir a verificação se a multiplicação está promovendo resultados corretos. A multiplicação de duas matrizes *NxN* resulta em uma matriz *NxN* com todos os elementos iguais a *N*.

4.1.2 Implementação de Funções de Multiplicação

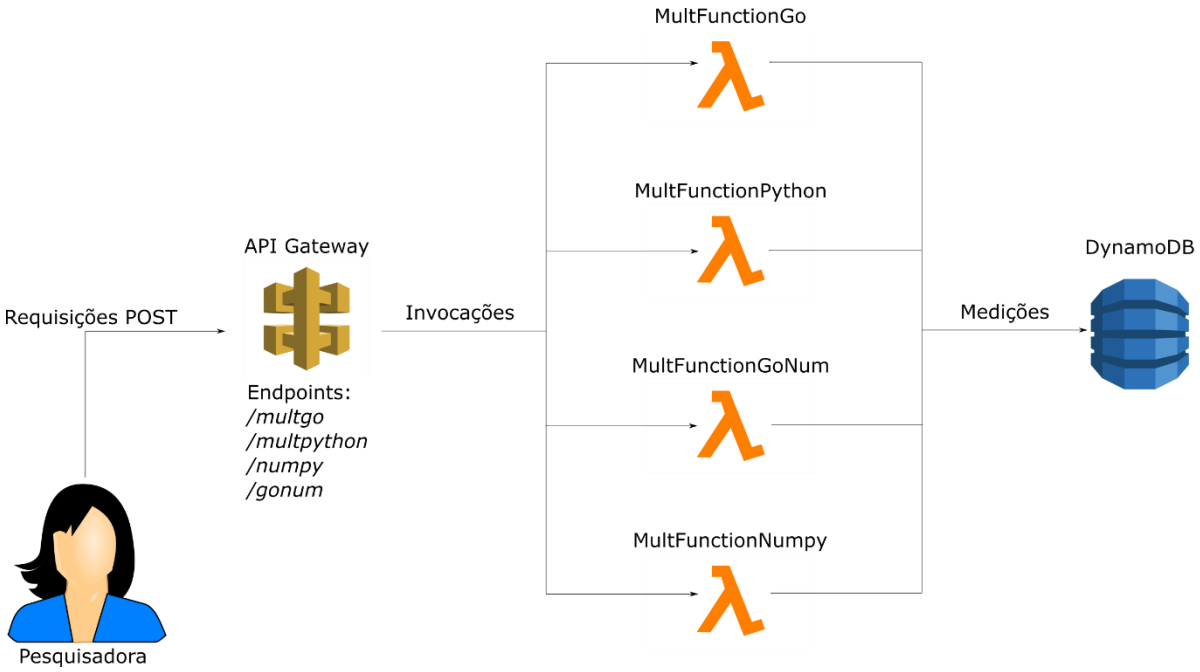
Após a conclusão da etapa anterior, implementamos as funções de multiplicação em ambas as linguagens. Essa etapa se refere ao nosso objetivo específico 3. Foram desenvolvidas funções que realizam a multiplicações de matrizes de maneira serial, uma versão em *Go* e outra em *Python*. Ainda nessa etapa configuramos o ambiente de execução das funções, como memória e duração de execução, para uma faixa de limites permitidos, nosso objetivo é identificar a dimensão máxima de matrizes que será possível multiplicar em cada linguagem de acordo com cada configuração de memória e tempo máximo de execução.

Dentre os trabalhos relacionados, citamos exemplos de soluções que paralelizam o próprio algoritmo de multiplicação através de *Step Functions*. Optamos por utilizar cada invocação de função responsável pela multiplicação serial de um par de matrizes por considerar que essa maneira é a mais adequada para isolar o impacto das linguagens na execução de cálculos científicos.

Para os experimentos, fizemos uso do SAM (*Serverless Application Model*)[8], uma solução da AWS para a automação de aplicações sem servidor. Este *framework* permite definir as funções e outros artefatos da nuvem em um modelo, um artefato que descreve a infraestrutura como código no formato YAML. Definimos no modelo uma API REST no serviço de *API Gateway*, as funções que implementam a multiplicação e uma base de dados no *DynamoDB* para armazenar informações sobre cada invocação de função, incluindo nesta o tempo de execução, qual linguagem foi utilizada e a ordem da matriz multiplicada.

[8] https://aws.amazon.com/pt/serverless/sam/

Figura 15 - Modelo SAM para Implantação do Ambiente de Testes.



Fonte: elaboração própria.

Na Figura 15, apresentamos os recursos criados pelo modelo SAM e como foram utilizados para realizar os experimentos. A pesquisadora realiza invocações HTTP POST na linha de comando utilizando a ferramenta *curl*, submetendo como parâmetro a ordem da matriz a ser multiplicada. De acordo com o recurso especificado na requisição, o *API Gateway* encaminha a requisição para a invocação de uma determinada função. Além das funções que implementam a multiplicação de matrizes de forma nativa na linguagem (*MultFunctionPython* e *MultFunctionGo*), temos as funções *MultFunctionGoNum* e *MultFunctionNumpy*, que utilizam as bibliotecas *NumPy*[9] e *GoNum*[10], respectivamente. A razão da inclusão dessas alternativas é explicada no próximo Capítulo, quando apresentamos os primeiros resultados.

4.1.3 Submissão de um Conjunto de Matrizes

Essa etapa consistiu na submissão de matrizes para as funções de multiplicação. Existem quatro variáveis para cada experimento:

(linguagem, ordem, memória, tempo máximo)

Por exemplo, uma submissão poderia ser configurada como *(python, 1000, 2GB, 6min)* para representar a execução da multiplicação de uma matriz 1000x1000 pela versão nativa do Python, utilizando 2 GB de memória RAM, utilizando um limite máximo de execução de 6

[9] https://numpy.org/
[10] https://www.gonum.org/

minutos. Essas duas últimas restrições são exigências do ambiente da *AWS Lambda*. O limite máximo de memória são 10 GB e o tempo máximo permitido são 15 minutos. Portanto, surge a questão de quais valores utilizar para avaliar as funções. Decidimos pela seguinte abordagem para a primeira etapa da análise:

1. Definir os intervalos de tempo máximos para uma das seguintes opções: **1min, 5min, 10min e 15min**.
2. Definir a configuração de memória para uma das seguintes opções: **2GB, 4GB, 6GB e 10 GB**.
3. Para cada par (intervalo, memória), definir uma opção de linguagem (*Python* ou *Go*) e incrementar a ordem da matriz até função reportar erro de estouro do limite de tempo ou memória.

Aplicando a metodologia acima, obtivemos o valor máximo da ordem de matriz que cada linguagem consegue computar em uma dada configuração de memória. Para a segunda etapa da análise:

1. Fixamos a memória em **6GB**.
2. Variamos o tempo máximo em **1min, 5min e 10min**.
3. Executamos a multiplicação submetendo a ordem de menor valor máximo obtido na primeira etapa com a mesma configuração de memória e tempo, em qualquer linguagem.
4. Repetimos o passo anterior 12 vezes, retirando o menor e o maior valor de tempo de execução, fazendo a média dos resultados restantes.

O passo 3 nos garante que qualquer uma das funções consegue executar o teste. Os resultados e discussão são matéria do próximo Capítulo.

5 RESULTADOS

Neste Capítulo apresentamos os resultados obtidos nas duas etapas de análise. Ao realizar a primeira etapa de análise descrita no Capítulo anterior obtivemos os resultados da *Tabela 2* na qual mostramos a definição das ordens limites e o tempo de execução (em *milissegundos*) das funções em *Python* e *Go* nos respectivos pares de configurações (memória, tempo limite).

Tabela 2 - Definição de Limites para Python e Go.

Tempo Limite	Memória	Ordem (Python)	Ordem (Go)	Tempo de Execução (Python)	Tempo de Execução (Go)
1min	2GB	611	1896	59922ms	58.734ms
1min	4GB	611	1896	59934ms	56.163ms
1min	6GB	608	1896	57667ms	55.846ms
1min	10GB	608	1896	59468ms	55.959ms
5min	2GB	1043	3120	298066ms	300.011ms
5min	4GB	1043	3120	294706ms	298.435ms
5min	6GB	1043	3120	295268ms	298.400ms
5min	10GB	1043	3120	287994ms	298.445ms
10min	2GB	1298	3800	585614ms	591.659ms
10min	4GB	1298	3800	579949ms	591.787ms
10min	6GB	1298	3790	573037ms	598.798ms
10min	10GB	1298	3801	579681ms	597.135ms
15min	2GB	1490	4229	869643ms	881.296ms
15min	4GB	1490	4228	869395ms	891.173ms
15min	6GB	1490	4235	886074ms	890.658ms
15min	10GB	1490	4236	873502ms	886.302ms

Fonte: elaboração própria.

Após obtenção destes resultados foi observado uma diferença de uma ordem de grandeza entre as ordens limites calculados nas duas linguagens. Como mostrado na *Tabela 2* a linguagem *Python* não obteve desempenho próximo ao de *Go,* já que as ordens máximas calculadas pela função *Python* possuem grande diferença com as de *Go* nas respectivas configurações, *Go* conseguiu multiplicar ordens maiores, em média o triplo da ordem que *Python* atingiu na mesma configuração. Partindo disso foi pensado em uma maneira de poder otimizar cada função utilizando uma biblioteca numérica em cada linguagem, com a utilização destas bibliotecas as funções poderiam vir a diminuir a diferença de desempenho.

Para a construção das novas funções otimizadas foram utilizadas as bibliotecas *NumPy* do *Python* e a *GoNum* de *Go*, ambas foram utilizadas para a construção das matrizes iniciais e para a multiplicação destas. A primeira etapa de análise também foi realizada para as duas novas funções e obtivemos os resultados contidos na *Tabela 3.*

Tabela 3 - Definição de Limites para NumPy e GoNum.

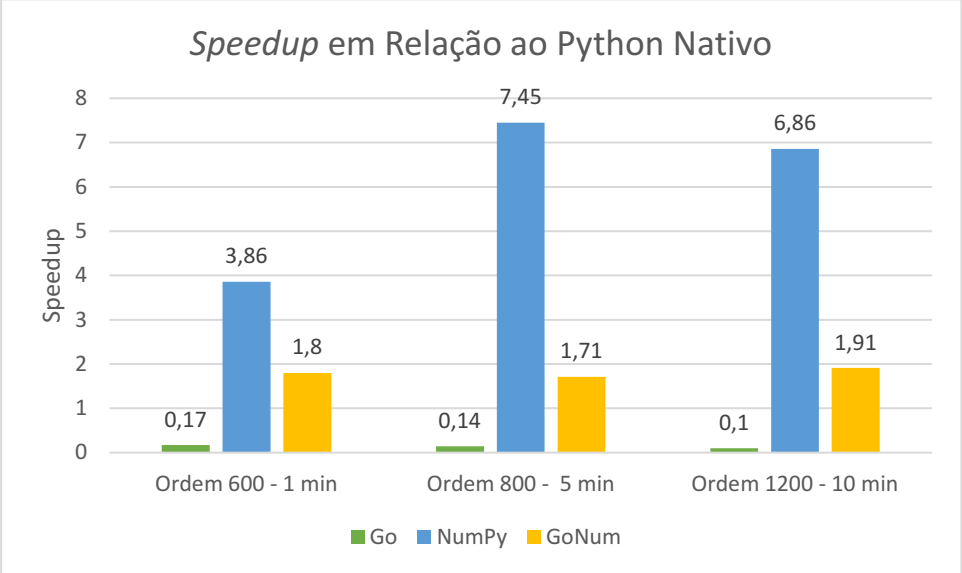
Tempo Limite	Memória	Ordem (NumPy)	Ordem (GoNum)	Tempo de Execução (NumPy)	Tempo de Execução (GoNum)
1min	2GB	9310	5020	37467ms	55.332ms
1min	4GB	13235	6290	53506ms	55.384ms
1min	6GB	15495	7200	59641ms	51.874ms
1min	10GB	18300	8210	56735ms	50.354ms
5min	2GB	9322	6565	43090ms	131.468ms
5min	4GB	13249	9389	55181ms	192.759ms
5min	6GB	16265	11900	75180ms	28.695ms
5min	10GB	21046	14090	91730ms	259.264ms
10min	2GB	9322	6565	44579ms	131.031ms
10min	4GB	13259	9400	64325ms	183.683ms
10min	6GB	16270	11540	67100ms	240.197ms
10min	10GB	21046	14700	91871ms	303.100ms
15min	2GB	9322	6565	45911ms	134.277ms
15min	4GB	13259	9400	62932ms	195.491ms
15min	6GB	16270	11500	79052ms	238.538ms
15min	10GB	21046	14710	90589ms	304.538ms

Fonte: elaboração própria.

Na *Tabela 3* vemos o quanto ambas as funções melhoraram seu desempenho, já que as ordens máximas calculadas por ambas aumentaram muito, trabalhando com as bibliotecas. Nestes resultados o *Python* com a biblioteca *NumPy* consegue realizar multiplicações para ordens maiores.

Continuando as etapas de análise, foi realizada a segunda etapa de análise descrita no Capítulo anterior. E assim como a primeira etapa, foi feita para as quatro funções, mostramos no gráfico abaixo os resultados para esta etapa da análise, este mostra o *Speedup* em Relação ao *Python* Nativo. O *speedup* é a divisão do tempo obtido pela *Python* pura sob o valor da solução em questão, ou seja, para calcularmos estes valores utilizamos os tempos de execução em *Python* pura e dividimos pôr os tempos de execução de *Go*, *NumPy* e *GoNum* nas respectivas ordens e tempo limite. Para uma melhor visualização dos dados os valores do gráfico estão em notação de ordem grandeza de 10^3.

Gráfico 1 - Speedup em Relação ao Python Nativo.



Fonte: elaboração própria.

Através destes resultados mostrados no gráfico, é possível concluir que entre as três alternativas o *Python* com a utilização da biblioteca *NumPy* obteve a melhor performance de tempo de execução, o *Go* com a *GoNum* vem em seguida como a segunda melhor alternativa e o *Go* puro segue como a terceira melhor. *NumPy* apresenta esses melhores resultados por seu funcionamento acabar invocando bibliotecas em C, mas, é importante destacar que a linguagem C não está disponível para construção de funções *lambda* da AWS.

Concluímos que a melhor alternativa para computação de alto desempenho no paradigma *Serverless* na plataforma AWS é a linguagem *Python* com a utilização da biblioteca *NumPy*, nossos experimentos comprovam a utilização desta no AWS *Lambda* além de comprovar um melhor desempenho em comparação com *Go* pura e *Go com* a biblioteca *GoNum*. Já para a execução pura entre *Python* e *Go* a melhor alternativa é *Go*, por ter obtido um melhor desempenho se comparada com *Python* pura.

6 CONCLUSÃO

A Computação Sem Servidor vem transformando o modo clássico de desenvolver aplicações de alto desempenho à medida que o desenvolvedor passa a não se preocupar com a infraestrutura que sua aplicação será executada. A utilização dos dois principais requisitos não funcionais da nuvem, escalabilidade e elasticidade, deixa o ambiente propicio para a execução dessas aplicações de alto desempenho. Este trabalho teve como principal objetivo analisar o desempenho destas aplicações no ambiente *serverless* na plataforma da nuvem AWS, através da comparação de duas linguagens, *Python* e *Go*, com a execução de funções de Multiplicação de Matrizes, problema bastante utilizado em diversos trabalhos.

Essa análise passa a ser de suma importância para desenvolvedores de aplicações de alto desempenho que se interessam em rodar suas aplicações no paradigma serverless. Saber qual a melhor linguagem deve utilizar, ainda era uma questão em aberto levando em conta o tipo de aplicação e o paradigma serverless. Neste trabalho obtivemos resultados satisfatórios em relação a estas questões. Através dos experimentos realizados foi possível constatar que entre as linguagens escolhidas para comparação, *Python* e *Go*, a melhor alternativa seria a utilização do *Python* com a biblioteca *NumPy*. Para experimentos entre funções com as duas opções de linguagens puras, a melhor alternativa é Go. Ambos os resultados obtiveram bons desempenhos comparados com suas funções semelhantes na linguagem contrária, estas conclusões foram obtidas analisando os limites de ordens das matrizes multiplicadas que cada função conseguiu executar, além de seus tempos de execução.

É esperado que estes resultados sejam de cunho importante para o público alvo, quanto para a comunidade em geral, assim como obteve papel importante no crescimento intelectual da autora deste trabalho. Para trabalhos futuros poderemos fazer esta análise abrangendo outras plataformas gerenciadoras de serviços na nuvem, como também outras linguagens devem ser selecionadas, já que cada plataforma possui suporte a linguagens diferentes que possam ou não estar presentes na plataforma utilizada neste trabalho. Acreditamos que ao realizar essa expansão de plataformas utilizadas nos experimentos possa trazer não somente resultados de desempenho das linguagens, como também trazer análise de custos entre diferentes plataformas.

REFERÊNCIAS

PETIT, A. et al. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. **HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**, [S.I], 2018. Disponivel em: http://www.netlib.org/benchmark/hpl/. Acesso em: 03 dez. 2020.

ANDREWS, G. R. **Foundations of Multithreaded, Parallel, and Distributed Programming**. [S.l.]: Addison-Wesley, 2000.

AWS. Computação de alta performance. **AWS**, [S.I], 2020a. Disponivel em: https://aws.amazon.com/pt/hpc/. Acesso em: 7 jul. 2020.

AWS. Implante um cluster elástico de HPC. **AWS**, [S.I], 2020b. Disponivel em: https://aws.amazon.com/pt/getting-started/hands-on/deploy-elastic-hpc-cluster/. Acesso em: 7 Jul. 2020.

AWS. Sem servidor. **AWS**, [S.I], 2020c. Disponivel em: https://aws.amazon.com/pt/serverless/. Acesso em: 7 Jul. 2020.

CHANIOTIS, I. K.; KYRIAKOU, K.-I. D.; TSELIKAS, N. D. Is Node.js a viable option for building modern web applications? A performance evaluation study. **Computer**, [S.I], v. 97, p. 1023–1044, out., 2015. Disponivel em: https://doi.org/10.1007/s00607-014-0394-9. Acesso em: 23 jul. 2020.

CLYNCH, D. J. E. G. **An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions**. IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). Zurich: IEEE. 2018. p. 154-160.

DONGARRA, J. **Visit to the National University for Defense Technology Changsha China. Visit to the National University for Defense Technology Changsha China**. Changsha: The National University for Defense Technology Changsha China, 2013. Disponivel em: http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf. Acesso em: 03 dez. 2020.

JONAS, E. et al. Cloud Programming Simplified: A Berkeley view on serverless computing. [S.l.]: **CoRR**. arXiv preprint arXiv:1902.03383. 2019. p. 1–33.

KOHN, S. **O que é e para quem é indicado a Serverless Computing**. [S.I]: Canaltech, 2018. Disponivel em: https://canaltech.com.br/infra/o-que-e-e-para-quem-e-indicado-a-serverless-computing-119782/. Acesso em: 18 maio. 2020.

MAGALHÃES, D. **Arquitetura de referência sem servidor: aplicativo da Web.** [S.I]: Repositório no GitHub, 2020. Disponivel em: https://github.com/aws-samples/lambda-refarch-webapp. Acesso em: 20 jul. 2020.

MATTSON G., T.; SANDERS A., B.; MASSINGILL L., B. **Patterns for Parallel Programming**. 1. ed. Boston : Pearson Education, 2004.

MELL, P.; GRANCE, T. **The NIST Definition of Cloud Computing**. [S.l.]: NIST, 2011.

MEYERSON, J. The Go Programming Language. **IEEE Software**, v. 31, p. 104 - 104, 2014. Disponivel em: https://ieeexplore.ieee.org/abstract/document/6898707. Acesso em: 23 jul. 2020.

OLIPHANT, T. E. Python for Scientific Computing. [S.I]: **Computing in Science & Engineering**, v. 9, p. 10 - 20, 2007. Disponivel em: https://ieeexplore.ieee.org/abstract/document/4160250. Acesso em: 23 jul. 2020.

PIPER, B.; CLINTON, D. **AWS Certified Cloud Practitioner Study Guide**. [S.l.]: Simbex, 2019.

POCCIA, D. **AWS Lambda in Action - Even-Driven Serverless Aplication**. Shelter Island, NY: Manning Publications Co., 2017.

ROSADO, T.; BERNARDINO, J. An Overview of Openstack Architecture. **Proceedings of the 18th International Database Engineering & Applications Symposium**, New York, NY, USA, 2014. 366–367. Disponivel em: https://doi.org/10.1145/2628194.2628195. Acesso em: 23 Jul. 2020.

SHANKAR, V. et al. **numpywren:** Serverless linear algebra. [S.l.]: arXiv preprint arXiv:1810.09679. 23 Out. 2018.

SPILLNER, J.; MATEOS, C.; MONGE, D. A. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. Cham: **Communications in Computer and Information Science**, p.154-168, v. 796, 2017.

STERLING, T.; ANDERSON, M.; BRODOWICZ, M. Chapter 1 - Introduction. In: _____ **High Performance Computing**. [S.l.]: Morgan Kaufmann, 2018. p. 1-42.

WERNER, S. et al. **Serverless Big Data Processing using Matrix Multiplication as Example**. IEEE International Conference on Big Data. Seattle, WA, USA: IEEE. 2018. p. 358–365.