UMA ANÁLISE DE PERFORMANCE DAS LINGUAGENS DE PROGRAMAÇÃO NO PROCESSAMENTO PARALELO

ARTIGO ORIGINAL

SOUZA, Robertson Douglas Castro[1], FLORIAN, Fabiana[2], DALLILO, Felipe Diniz[3]

SOUZA, Robertson Douglas Castro. FLORIAN, Fabiana. DALLILO, Felipe Diniz. **Uma análise de performance das linguagens de programação no processamento paralelo.** Revista Científica Multidisciplinar Núcleo do Conhecimento. Ano 08, Ed. 12, Vol. 04, pp. 05-37. Dezembro de 2023. ISSN: 2448-0959, Link de acesso: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao, DOI: 10.32749/nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

RESUMO

Cada linguagem é otimizada para resolver determinado tipo de tarefa, portanto, é necessário ter conhecimento sobre suas limitações e pontos fortes para escolher a que melhor atenderá à necessidade. O paralelismo é uma técnica que tem se destacado como fundamental em razão da crescente complexidade dos problemas computacionais. Este trabalho visa analisar a performance da execução e a implementação de atividades, neste cenário. A partir de estudos realizados com algumas linguagens que se destacam atualmente no mercado para a construção de aplicações backend, e que avaliaram a complexidade para desenvolvimento da solução e o aproveitamento dos recursos computacionais, realizou-se um benchmarking de performance entre: GO, JAVA, C# e PYTHON. Os resultados apontam que as linguagens GO e C# se destacam de maneira relevante perante as outras no paralelismo, tornando-as melhores opções entre as demais para a tarefa.

Palavras-chave: Linguagens de Programação, Paralelismo, *Multicore*, Benchmark, C# e .NET, Go.

1 INTRODUÇÃO

Com o avanço da tecnologia e cada vez mais a facilidade ao seu acesso, tornou-se comum o uso de aplicativos e demais sistemas para auxiliar pessoas em suas atividades diárias. Essa facilidade do uso da tecnologia traz novos desafios aos profissionais que criam os sistemas como, por exemplo, garantir disponibilidade aos usuários em picos de acesso, como também em alguns outros cenários, preocupar-se em garantir rápidas respostas diante das solicitações.

Para que as novas, ou até mesmo as antigas aplicações, continuem atendendo às necessidades dos usuários, as tecnologias precisam ser adaptadas a elas. Os sistemas e as tecnologias utilizadas em suas construções precisaram se tornar mais inteligentes, otimizando a utilização dos recursos disponíveis pelas máquinas, tanto por uma questão de custo, quanto de performance.

No campo da performance, com os processadores *multicore*, ou em português multinúcleo, pode-se explorar o cenário do processamento em paralelo, que Hwang (1984, *apud* Navaux, 1988, p. 1) explicou como "uma forma eficiente do processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional". O uso dessa estratégia traz maior complexidade para as soluções a serem desenvolvidas, exige maior conhecimento técnico dos profissionais pela necessidade de um melhor controle e aproveitamento dos recursos computacionais utilizados durante a execução de um programa.

Por conta dessa necessidade, algumas linguagens de programação aprimoraram sua forma de trabalho com o paralelismo e algumas outras estratégias para otimizar sua execução: evoluíram a forma de escrever o código, trouxeram simplicidade para o desenvolvimento em alguns cenários e potencializaram a sua forma de trabalhar.

Este artigo tem o objetivo de analisar a performance da execução e da implementação de algoritmos para processar uma atividade em paralelo, com base

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

em algumas das linguagens que se destacam atualmente no mercado para a construção de aplicações backend.

Foi realizada uma pesquisa bibliográfica e utilizou-se de um computador pessoal multicore para implementar os algoritmos que foram executados e tiveram suas performances avaliadas. Considerou-se o tempo total que levaram para realizar a mesma atividade e o uso dos recursos disponibilizados pela máquina.

Todos os algoritmos foram executados nas mesmas condições e um determinado número de vezes, para gerarmos uma média a ser trabalhada, em busca de uma melhor precisão no resultado. Todos os programas não necessários fechados e os necessários já abertos antes das execuções. Estudou-se as seguintes linguagens de programação: Go, Python, Java, e C#.

2 REVISÃO BIBLIOGRÁFICA

2.1 COMPUTADORES MULTICORES

Devido ao aumento da complexidade na criação dos *softwares* e a necessidade de otimização do uso dos recursos computacionais, os computadores multicores trouxeram uma reestruturação do modo de trabalhar no meio computacional permitindo contornar os limites de performance encontrados na época.

Como a maioria das abordagens estava em seu limite, era necessária uma nova técnica para aumentar a velocidade de processamento, aumentar a frequência ou o número de transistores nos processadores não atendia mais o problema. Uma nova direção foi necessária e está vem justamente ao seguir o caminho inverso das abordagens utilizadas até os dias atuais. Em lugar de aumentar a frequência dos processadores, a solução foi diminuir a frequência de seu núcleo e aumentar o número destes em cada processador, podendo gerar ganhos de 30 a 90 por cento

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

sem aumentar, na mesma proporção, a geração de calor e o consumo de energia (Brandel, 2004, *apud* Ma, [s.d.]).

Anthes (2004, apud Ma, [s.d.]) observa que, em geral, processadores multinúcleos são definidos como sendo chips com mais de uma unidade de processamento, cada uma com capacidade para executar múltiplos processos por vez, ou, um circuito integrado com dois ou mais processadores para aumentar sua performance, com baixa utilização de energia e que executa com mais eficiência múltiplos processos simultâneos.

Ma (s.d., n. p.) explica que a tecnologia dominante atualmente permite um socket proveja acesso a um núcleo lógico, proporção de 1 para 1, já no caso das novas abordagens criadas para esses novos processadores permitem para cada socket prover acesso para dois, quatro ou mais núcleos presentes em um único processador. Traz o ponto de forma simplificada de Intel em *Intel Multi-Core Processor Architecture Development Backgrounder*:

Um processador multinúcleo funciona como se, ao invés de um processador, haja em seu computador dois ou mais processadores trabalhando em conjunto em um mesmo socket, o que viabiliza que a conexão entre eles se realize mais rapidamente do que em sistemas multiprocessados onde são utilizados um socket por processador (Ma, [s.d.], n. p.).

2.2 PARALELISMO EM COMPUTADORES MULTICORES

Com a evolução dos processadores para um contexto *multicores*, novas necessidades surgiram no campo dos *softwares*, passaram a existir novos desafios e algoritmos mais complexos.

Enquanto os processadores apenas aumentavam sua frequência para melhorar a velocidade de processamento, a indústria do *software* apenas precisava realizar

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

pequenas mudanças nos códigos e ver sua performance aumentar assim que as frequências eram potencializadas. O projeto de múltiplos núcleos está forçando o mundo dos *softwares*, de modo geral, o que inclui tanto clientes quanto desenvolvedores, a resolver com computação paralela, concorrente, capacitando um programa em se dividir em muitas partes pequenas que sejam trabalhadas separadamente e, depois, recombinadas. (Ma, [s.d.], n. p.).

Ma (s.d., n. p.) acredita em uma vantagem da tecnologia a longo prazo: [...] os processadores multinúcleo irão desacelerar o ritmo de depreciação do "*hardware*" pois estes têm a capacidade de executar os programas atuais e estão preparados para executar programas de um futuro próximo. A crescente complexidade dos programas combinados com o desejo de executar mais de um programa por vez irá acelerar o processo de disseminação destes novos modelos de processadores. Além disso, os programadores estão mudando seus paradigmas de programação, iniciando uma era de programação mais paralela, que aproveitaria o máximo desses novos processadores.

Também se nota que a nova estratégia tecnológica adotada afetou todo ecossistema computacional, desde as máquinas, programadores, até as próprias linguagens e ferramentas que auxiliam as tarefas do programador. Conforme explica Scapin (2013, p. 13):

> Tradicionalmente, *softwares* são escritos para utilizar processamento sequencial. As máquinas continham uma única unidade central de processamento (CPU), fazendo com que as instruções fossem executadas uma após a outra, individualmente. Com os avanços na área da computação, é possível utilizar uma máquina com múltiplas unidades centrais de processamento para usufruir do processamento paralelo.
>
> O processamento paralelo é o uso simultâneo de múltiplos recursos computacionais para resolver um problema (Barney, 2012). Diferente das máquinas com uma única CPU, as que possuem múltiplas CPU ou núcleos podem dividir um problema grande em partes menores para executá-los

> paralelamente e/ou concorrentemente. Dessa maneira, é possível obter melhorias em tempo de processamento, sobrecarga em *hardware*, dividir problemas grandes para ser resolvido em partes, utilização de recursos externos, entre outros.

A estratégia não traz só benefícios, em contrapartida, aumenta a complexidade dos códigos a serem desenvolvidos, como complementa Sakuray (2014, p. 40):

> Tarefas que são executadas simultaneamente adicionam detalhes que não aparecem na computação sequencial. A dependência de um mesmo recurso por várias tarefas que são executadas em paralelo é um exemplo onde é necessário maior atenção e cuidado.

Ainda por Scapin (2013, p. 13-14), explica:

> A tarefa de escrever algoritmos de paralelização apresenta desafios, que são considerados mais difíceis que aqueles encontrados em algoritmos sequenciais. Em geral, a paralelização é completamente tarefa do programador. Para auxiliar nesse desenvolvimento, existem ferramentas e bibliotecas que contém recursos para o programador utilizar. Algoritmos paralelos são largamente utilizados para processamento de imagens digitais, processamento de gráficos computacionais, previsões meteorológicas em tempo real, tarefas de video *encoding,* entre outras.

Mesmo que a estratégia de processamento de dados em paralelo continuamente apresenta desafios, os avanços contínuos das tecnologias e algoritmos permitem quebrar essas barreiras e trabalhar com grandes volumes de dados de forma eficiente e escalável.

Pode-se observar que o cenário do paralelismo trouxe muitos benefícios, mas com alguns custos, principalmente no aumento da complexibilidade, e para isso tecnologias também evoluíram e se adaptaram para auxiliarem no trabalho dos desenvolvedores.

2.3 LINGUAGENS DE PROGRAMAÇÃO

Conforme Gotardo (2015, p. 17) explica:

> É um método padronizado que usamos para expressar as instruções de um programa a um computador programável. Ela segue um conjunto de regras sintáticas e semânticas para definir um programa de computador. Regras sintáticas dizem respeito à forma de escrita e regras semânticas ao conteúdo.
>
> Através da especificação de uma linguagem de programação é possível especificar quais dados um computador utiliza; como estes dados serão tratados, armazenados, transmitidos; quais ações devem ser tomadas em determinadas circunstâncias.
>
> Ao usar uma linguagem de programação cria-se o chamado "Código Fonte". Um código fonte é um conjunto de palavras escritas de acordo com as regras sintáticas e semânticas de uma linguagem.

Para este trabalho foi utilizado as linguagens de programação Java, Python, C# com .NET e Go, que mais se destacam em uso e popularidade no mercado atualmente.

2.3.1 JAVA

De acordo com as documentações oficiais da linguagem, explica-se que:

> É uma linguagem de programação e plataforma de computação, lançada pela primeira vez pela Sun Microsystems em 1995. Teve um início humilde e evoluiu para uma grande participação no mundo digital dos dias atuais, oferece uma plataforma confiável na qual muitos serviços e aplicativos são desenvolvidos. Produtos e serviços novos e inovadores projetados para o futuro continuam a confiar no Java também (Java, [s.d]).

Desde seu lançamento continuou evoluindo e atualmente já alcançou sua 20ª versão, seguindo a linha apresentada pelo estudo, que mostra a necessidade das tecnologias se adaptarem às novas demandas de mercado (Oracle, [s.d.]).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

2.3.2 PYTHON

O Python apresenta diversos benefícios que facilitam o trabalho do desenvolvedor no dia a dia, como uma sintaxe básica semelhante à do inglês, que permite facilidade na leitura e escrita dos códigos, uma grande biblioteca-padrão que contém códigos reutilizáveis para quase todas as tarefas, resultando que os desenvolvedores não precisam frequentemente escrever códigos do zero, uma forte comunidade e diversos outros pontos (Amazon, [s.d.]).

> Possui estruturas de dados eficientes de alto nível e uma abordagem simples, mas eficaz, para programação orientada a objetos. Sua sintaxe elegante e a digitação dinâmica, juntamente com a sua natureza interpretada, o torna uma linguagem ideal para scripts e desenvolvimento rápido de aplicativos em muitas áreas na maioria das plataformas (Python, [s.d.]).

2.3.3 C# E .NET

Conforme material disponibilizado pela Microsoft, empresa que fornece o suporte para as tecnologias C# e .NET, explica-se que:

> O .NET é uma plataforma de desenvolvedor multiplataforma de código aberto gratuita para criar muitos tipos diferentes de aplicativos, com ele pode se usar vários idiomas, editores e bibliotecas para criar para Web, dispositivos móveis, desktop, jogos, IoT e muito mais. Permite a escrita de aplicativos nas linguagens C#, F# ou Visual Basic.
>
> O C# (pronuncia-se "C Sharp") é uma linguagem de programação moderna, orientada a objeto e fortemente tipada. Permite que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados no .NET. Tem suas raízes na família de linguagens C (Microsoft, [s.d.]a).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

2.3.4 GO

Segundo Rob Pike (2007), mesmo com a evolução constante das tecnologias já existentes, ainda existia a necessidade de algo mais performático no mercado e que fosse mais fácil de se trabalhar para auxiliar a solucionar os desafios diários de engenharia.

A linguagem foi criada pelo Google em 2007 e disponibilizada ao público em 2009, visando melhorar a produtividade da programação em uma era de máquinas em rede com vários núcleos e grandes bases de código (Microsoft, [s.d.]b). É construída em cima de ideias de outras linguagens já existentes, busca entregar o máximo de *performance* e otimizar o uso de recursos de máquinas multicores e em rede.

2.4 TRABALHOS SIMILARES

Com base nos trabalhos apresentados por David *et al.* (2022), Östlund (2016) e Sundar (2019), e a forma que conduziram as avaliações entre as linguagens, concluiu-se que a análise do tempo de execução das funções em relação ao uso de CPU e memória da máquina, mais a complexidade dos algoritmos em cada linguagem, já fornecem insumos suficientes para que o estudo proposto atenda seu objetivo.

Dadas as diversas atividades que podem ser executadas em um algoritmo, com os parâmetros citados anteriormente, foi possível tirar conclusões de maneira prática de quais linguagens atendem melhor o cenário de concorrência e como lidam com os recursos disponibilizados pelas máquinas.

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

2.5 PARÂMETROS AVALIADOS

Foram os escolhidos os seguintes parâmetros para avaliação da performance dos algoritmos nas linguagens: média dos usos da memória RAM[4] e CPU[5], tempo gasto medido em milissegundos, e análise de complexidade de cada uma.

O uso de recurso de memória RAM e CPU foi medido com o auxílio da ferramenta do próprio sistema operacional da máquina, no caso, o gerenciador de tarefas do Windows. Para o tempo de execução, a tarefa de interesse a ser avaliada nas diferentes linguagens teve o tempo gasto por recursos nativos e escritos no console.

> E a complexidade de algoritmos, que consiste na quantidade de trabalho necessária para execução, expressa em função das operações fundamentais, que variam de acordo com o algoritmo em função do volume de dados. As operações fundamentais são aquelas que, dentre as demais operações que compõem o algoritmo, expressam a quantidade de trabalho, portanto dependem totalmente do problema e do algoritmo (Barbosa e Toscani, 2022, n. p.).

Complementa Rosa (2021, n. p.), em sua obra:

> A notação Big O é uma das ferramentas mais importantes para os cientistas da computação analisarem o custo de um algoritmo. Dito de modo um pouco mais simples, a notação descreve a complexidade do seu código usando termos algébricos.

2.6 REPOSITÓRIOS REMOTOS

Por conta do foco do estudo, pontos que talvez interessem ao leitor podem não ter sido abordados, ou simplesmente, caso haja um interesse de olhar mais detalhadamente os algoritmos trabalhados, e até executá-los, está disponibilizada, abaixo, a lista com os devidos repositórios remotos que contém orientações para a execução.

- Script gerador da amostra de dados e arquivo .yaml com a configuração da base - https://github.com/robert-dcs/parallelExecutionSample
- GO - https://github.com/robert-dcs/goParallelProcessing.
- C# - https://github.com/robert-dcs/.NetParallelProcessing.
- JAVA - https://github.com/robert-dcs/javaParallelProcessing.
- PYTHON - https://github.com/robert-dcs/pythonParallelProcessing.

3. DESENVOLVIMENTO

A implementação dos códigos foi feita com o objetivo de utilizar apenas os recursos nativos das linguagens para a implementação do processamento em paralelo, e evitar o uso de frameworks para, assim, ser o mais justo possível na comparação.

O foco deste trabalho está na análise da *performance* da execução de algoritmos que processam conjuntos de dados em paralelo, de como as linguagens usam os recursos computacionais disponíveis para tal tarefa e, também, em entender quanto da complexidade desses cenários as linguagens abstraem para o programador.

3.1 AMBIENTE DE DESENVOLVIMENTO

Para o desenvolvimento, foi utilizado um computador pessoal *multicore* com as configurações descritas conforme tabela 1 e as versões das linguagens que foram detalhadas conforme a tabela 2.

Tabela 1: Informações do *Hardware*

Placa mãe	**Placa Mãe Asus TUF Gaming B550M-Plus DDR4**
CPU	**AMD Ryzen 5 5600X 6-Core Processor 3.70 GHz**
GPU	**AMD Radeon RX 6600 XT 8GB GDDR6 128 Bit**
RAM	**16 GB 2333 MHz DDR4**
Sistema Operacional	**Windows 10 Pro Versão 22H2 x64**

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Fonte: Os autores (2023).

Tabela 2: Informações das Linguagens

Java	**jdk-16.0.2**
Go	**go 1.20.3.windows-arm64.msi**
Python	**3.11**
C# e .NET	**C# 11 e .NET 7.0**

Fonte: Os autores (2023).

Inicialmente, foram testadas algumas possíveis abordagens diferentes para os algoritmos nas linguagens, porém a fim de comparação com as demais linguagens, utilizou-se da abordagem mais performática, escolha feita ao comparar os tempos de execução e analisar os algoritmos que mais se destacavam, comparação está feita com a utilização das mesmas amostras do benchmarking principal.

3.3 PRINCIPAL DESAFIO DA IMPLEMENTAÇÃO PARA ESTE TRABALHO

Se tratando do paralelismo, o principal desafio foi a concorrência do uso do recurso da base de dados. Não foi criada nenhuma solução extremamente trabalhada, e sim sanados os problemas necessários para que o código pudesse ser executado e o trabalho pudesse ser realizado.

A mínima estratégia requerida por todas as linguagens foi o uso de um pool de conexão com o banco de dados, o que viabilizou que todas as operações fossem executadas em paralelo. No caso do C#, a biblioteca utilizada já realizava a abstração do uso do pool de conexões com a base, sem se fazer necessário alguma configuração a respeito, o que não foi o caso das demais linguagens.

O foco do trabalho não está nos detalhes da configuração com a base de dados implementada em cada linguagem, mais detalhes que podem ser consultados nos

códigos fontes, mas sim, em avaliar as ferramentas que as linguagens provêm para se implementar um processamento em paralelo, bem como a facilidade que as mesmas conferem ao desenvolvedor para fazer essa implementação, e a performance desses algoritmos.

3.4 IMPLEMENTAÇÃO DO ALGORITMO COM GO

A linguagem Go se apresentou simples de trabalhar, e preparar seu ambiente utilizando o Intellij para o trabalho, fez-se necessário apenas baixar os recursos da linguagem e instalar o plug-in que dá suporte para a IDE.

Um dos outros desafios que o desenvolvedor possui em algumas linguagens, é trabalhar com pacotes externos, no Go se mostrou extremamente tranquilo, seu gerenciador nativo de dependência é totalmente simples de trabalhar, no caso foi utilizado apenas uma biblioteca para leitura de arquivos .xlsx, de onde foram tiradas as amostras de dados, e uma biblioteca para comunicação com o banco de dados.

No caso da linguagem, o que auxiliou no processamento paralelo foi o uso da palavra reservada go antes da função em que chamamos a persistência dos dados no banco, o que faz com que, durante a execução do código, o laço promova suas repetições em goroutines separadas, executadas concorrentemente, e isso possibilita a atuação em paralelo nesse contexto. As diferenças podem ser observadas entre os quadros 1 e 2, comparativamente:

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 1: Algoritmo de processamento paralelo em Go

```
import (
	"context"
	"fmt"
	"github.com/xuri/excelize/v2"
	"strconv"
	"sync"
	"time"
)

func parallelProcessing(ctx context.Context, listOfPeople []string) {
	var wg sync.WaitGroup
	personDb := newConnection(ctx)
	defer personDb.dbConn.Close()
	personDb.dropAndCreateDatabase(ctx)
	count := 0

	start := time.Now()
	for _, person := range listOfPeople {
		wg.Add(1)
		personP := person
		go func() {
			err := personDb.persistPerson(ctx, personP, &wg)
			if err != nil {
				fmt.Println(err)
			}
		}()
		if count >= 2000 {
			wg.Wait()
			count = 0
		}
		count++
	}
	wg.Wait()
	elapsed := time.Since(start).Milliseconds()

	fmt.Printf("[Go] Parallel implementation 1 tooked %d milliseconds.\n", elapsed)
	fmt.Printf("[Go] Processed %d records.\n", len(listOfPeople))
}
```

Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 2: Algoritmo de processamento sequencial em Go

```
import (
	"context"
	"fmt"
	"github.com/xuri/excelize/v2"
	"strconv"
	"sync"
	"time"
)


func synchronousProcessing(ctx context.Context, listOfPeople []string) {
	personDb := newConnection(ctx)
	personDb.dropAndCreateDatabase(ctx)
	start := time.Now()
	for _, person := range listOfPeople {
		dbError := personDb.persistPerson(ctx, person, nil)
		if dbError != nil {
			panic(dbError)
		}
	}
	elapsed := time.Since(start).Milliseconds()
	fmt.Printf("[Go] Parallel implementation tooked %d milliseconds.\n", elapsed)
	fmt.Printf("[Go] Processed %d records.\n", len(listOfPeople))
}
```

Fonte: Os autores (2023).

Se fez necessário uma estrutura de sincronização, pois uma vez criada, o código continuará a execução das próximas linhas, enquanto a *goroutine* executa o trecho de código de sua responsabilidade. Caso o restante do código termine antes da execução da *goroutine*, o programa será encerrado e não teremos o resultado da sua execução, por isso é necessário usar um mecanismo de sincronização para que o programa não encerre antes do desejado.

Para isso foi utilizado o pacote nativo *sync.WaitGroup*, com seu auxílio, de uma forma simples era comunicado que havia goroutines em execução e que o programa precisava espera-las terminar a execução para partir para a próxima ação.

Deste pacote, foram utilizados apenas 3 métodos para garantir a sincronização, dentro do laço de repetição. Utilizou-se: a) o método *Add()* para sinalizar ao programa que em cada laço havia uma *goroutine* em execução; b) ao término do for utilizou-se o *Wait()*, que garantiu a espera da finalização da *goroutine*, conforme pode ser observado no quadro 1; c) sinalizar o término da tarefa realizada dentro do método, para salvar na base, foi atribuição do método *Done()*, que está sendo executado pela *goroutine*, e demonstrado no quadro 3.

Quadro 3: Método de inserção no banco de dados

```
import (
        "context"
        "fmt"
        "github.com/jackc/pgx/v4/pgxpool"
        "sync"
)


func (c DbConnection) persistPerson(ctx context.Context, person string, wg
*sync.WaitGroup) error {
        if wg != nil {
                defer wg.Done()
        }

        sql := fmt.Sprintf("insert into person(name) values ('%s')", person)
        _, err := c.dbConn.Exec(ctx, sql)
        if err != nil {
                return err
        }

        return nil
}
```

Fonte: Os autores (2023).

O Go tem a proposta de simplificar a escrita do código, mas não necessariamente a de trazer abstrações. Na verdade, Go se assemelha a uma linguagem de mais

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

baixo nível, onde é comum se fazer algumas implementações um pouco mais trabalhadas, como, por exemplo, a do próprio controle do paralelismo, para o qual foi preciso o uso de um controle sobre as goroutines, como a sincronização com a execução do programa e controle do número de goroutines abertas, pois gerava concorrência na conexão com a base de dados, o que provocava erro com o aumento da carga de dados a serem processados.

Para isso, estudou-se um número possível de *goroutines* para se trabalhar e limitar as suas criações através de um contador durante a execução de grandes cargas. O trabalho pode parecer complexo, mas não precisou de muito tempo para que se obtivessem números que viabilizassem a execução da aplicação sem erros e com uma boa performance. O cenário para performance não foi explorado de modo privilegiado. Trabalhou-se apenas o suficiente para que não houvesse problemas de concorrência com o uso da base de dados.

3.5 IMPLEMENTAÇÃO DO ALGORITMO COM JAVA

otar que essa linguagem traz um nível de abstração maior na escrita do que a Go, em alguns pontos, como podemos observar no quadro 4, em relação ao algoritmo de processamento sequencial. Porém, para o paralelismo, não foi este o cenário mais simples de implementação.

Por conta da performance que os algoritmos mostraram, o algoritmo escolhido para seguir com as comparações foi o que utilizou a classe *CompletableFuture*. Com o auxílio da classe *ExecutorService*, definiu-se o número de *threads* em que se distribuíram as tarefas para execução assíncrona.

E com o método *CompletableFuture.allOf().join()*, foi feita a sincronização das tarefas que foram paralelizadas sem muita dificuldade, conforme se pode observar no quadro 5. Nota-se a necessidade de uma implementação um pouco mais

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

trabalhosa do que as abstrações fornecidas pela linguagem, para uma melhor performance.

Quadro 4: Algoritmo de Sequencial paralelo em Java

```
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.WorkbookFactory;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;

static void synchronousProcessing(List<String> listOfPeople) throws SQLException {
      DbConnection.cleanDatabase();
      long startTime = System.currentTimeMillis();
      listOfPeople.forEach(person -> {
         try {
            DbConnection.insertPerson(person);
         }catch (Exception e) {
            System.out.println(e.getMessage());
         }
      });
      long endTime = System.currentTimeMillis();
      long elapsedTime = endTime - startTime;
      System.out.println("[Java] Synchronous implementation took " + elapsedTime + "
milliseconds." );
      System.out.println("[Java] Processed " + listOfPeople.size() + " records.");
   }
```

Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 5: Algoritmo de processamento paralelo em Java

```
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.WorkbookFactory;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;

static void parallelProcessing2(List<String> listOfPeople) throws SQLException {
      DbConnection.cleanDatabase();
      ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
      long startTime = System.currentTimeMillis();
      try {
         List<CompletableFuture<Void>> futures = listOfPeople.stream()
               .map(person -> CompletableFuture.runAsync(() -> {
                  try {
                     DbConnection.insertPerson(person);
                  } catch (SQLException e) {
                     throw new RuntimeException(e);
                  }
               }, executor))
               .toList();

         CompletableFuture.allOf(futures.toArray(CompletableFuture[]::new)).join();
      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         executor.shutdown();
      }
      long endTime = System.currentTimeMillis();
      long elapsedTime = endTime - startTime;
      System.out.println("[Java] Parallel implementation 2 took " + elapsedTime + " milliseconds." );
      System.out.println("[Java] Processed " + listOfPeople.size() + " records.");
   }
```

Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

3.6 IMPLEMENTAÇÃO DO ALGORITMO COM C# E .NET

A implementação do algoritmo em C# com a plataforma .NET demonstrou-se uma linguagem extremamente tranquila para trabalhar e possui um gerenciador de dependência amigável que auxiliou de forma positiva o desenvolvimento do projeto.

Similar ao Java, pode se notar um maior nível de abstração provido pela linguagem em alguns recursos, assim como o mecanismo para o processamento em paralelo. Fez-se necessário apenas o uso de um método nativo que permite o processamento dos elementos de uma lista de forma paralela. Assim como é possível percorrer uma lista de forma simples com o método *foreach()*, conforme a quadro 6, pode-se fazer isso de forma paralela, ao utilizar o método *Parallel.ForEach()*, mostrado no quadro 7.

Dos algoritmos avaliados, essa abordagem da linguagem se mostrou a mais simples e performática. Possui uma performance similar à outra implementada, mas maior simplicidade na escrita. Porém, quando comparamos com o pacote de *Thread*, obtivemos as seguintes informações: a) performance muito maior; b) muita dificuldade para garantir sincronização da execução em relação ao *Parallel.ForEach()*.

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 6: Algoritmo de processamento sequencial em C#

```
using Npgsql;
using ClosedXML.Excel;

static void SynchronousProcessing(List<string> listOfPeople)
      {
         var dataBaseCon = new DBconnection.Database();
         using var con = dataBaseCon.getConnection();
         dataBaseCon.dropAndCreateTable();
         var watch = System.Diagnostics.Stopwatch.StartNew();
         foreach (string person in listOfPeople)
         {
            dataBaseCon.inserPersonToDB(person);
         };
         watch.Stop();

         var elapsedMs = watch.ElapsedMilliseconds;
         Console.WriteLine("[C#] Synchronous implementation took " + elapsedMs + "
milliseconds.");
         Console.WriteLine("[C#] Processed " + listOfPeople.Count() + " records.");
      }
```

Fonte: Os autores (2023).

Quadro 7: Algoritmo de processamento paralelo em C#

```
using Npgsql;
using ClosedXML.Excel;

 static void ParallelProcessing2(List<string> listOfPeople)
      {
         var dataBaseCon = new DBconnection.Database();
         using var con = dataBaseCon.getConnection();
         dataBaseCon.dropAndCreateTable();
         var watch = System.Diagnostics.Stopwatch.StartNew();
         Parallel.ForEach(listOfPeople, dataBaseCon.inserPersonToDB);
         watch.Stop();
         var elapsedMs = watch.ElapsedMilliseconds;
         Console.WriteLine("[C#] Parallel implementation 2 took " + elapsedMs + " milliseconds.");
         Console.WriteLine("[C#] Processed " + listOfPeople.Count() + " records.");
      }
```

Fonte: Os autores (2023).

No caso da biblioteca utilizada para conectar com a base de dados, até seu método básico de conexão já trazia consigo a abstração do uso de um pool de conexões, o que dispensava qualquer necessidade de preocupação do desenvolvedor quanto a isso, inclusive quanto à necessidade de realizar configurações. A única preocupação do desenvolvedor é a forma correta de se trabalhar com o objeto de conexão, que também não é de grande dificuldade.

3.7 IMPLEMENTAÇÃO DO ALGORITMO COM PYTHON

A preparação do ambiente para o Python utilizando o Intellij foi tranquila, houve apenas a necessidade de baixar os recursos da linguagem e instalar o plug-in que dá suporte ao Intellij, como em Go.

No caso dos pacotes externos, seu sistema nativo também é tranquilo de trabalhar. Houve necessidade apenas das mesmas bibliotecas dos demais, para leitura de arquivos .xlsx e para comunicação com o banco de dados.

Dados os algoritmos explorados, foi utilizada a classe *Thread* do pacote *threading*, que permite a implementação de um algoritmo que fornece a paralelização de tarefas um pouco mais performáticas que as demais. Não foi o algoritmo mais simples da linguagem para a tarefa, mas o cenário não exigiu uma estrutura muito elaborada. A implementação consistiu em: a) criar uma fila para inserir cada elemento a ser processado; b) fornecer uma lista de *threads* com a quantidade necessária ao trabalho; c) aplicar um método auxiliar que dá suporte para as *threads* com a estrutura de consumo dos dados da fila que serão distribuídos para o processamento.

Nos quadros 8 e 9 pode-se observar as diferenças do algoritmo sequencial e paralelo com Python, fazendo-se necessário apenas declarar a classe comentada e, para cada elemento da amostra, disparamos o método de processamento assíncrono que distribui a execução em *threads*.

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 8: Algoritmo de processamento sequencial em python

```
import threading
import openpyxl
import psycopg2
import psycopg2.extras
import time
import concurrent.futures
from psycopg2 import pool
from multiprocessing.dummy import Pool
import os
from queue import Queue

def synchronous_processing(listOfPeople):
    drop_and_create_table()
    start = (time.time() * 1000)
    for person in listOfPeople:
        insert_person(person)
    end = (time.time() * 1000)
    elapsed_time = end - start
    print("[Python] Synchronous implementation took " + str(elapsed_time) + " milliseconds.")
    print("[Python] Processed " + str(len(listOfPeople)) + " records.")
```

Fonte: Os autores.

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Quadro 9: Algoritmo de processamento paralelo em Python

```
import threading
import openpyxl
import psycopg2
import psycopg2.extras
import time
import concurrent.futures
from psycopg2 import pool
from multiprocessing.dummy import Pool
import os
from queue import Queue

def parallel_processing2(listOfPeople):
    MAX_THREADS = os.cpu_count()
    queue = Queue()

    for person in listOfPeople:
        queue.put(person)

    threads = []
    start = (time.time() * 1000)
    for _ in range(MAX_THREADS):
        thread = threading.Thread(target=worker, args=(queue,))
        thread.start()
        threads.append(thread)

    queue.join()

    for _ in range(MAX_THREADS):
        queue.put(None)
    for thread in threads:
        thread.join()

    end = (time.time() * 1000)
    elapsed_time = end - start
    print("[Python] Parallel implementation 2 took " + str(elapsed_time) + " milliseconds.")
    print("[Python] Processed " + str(len(listOfPeople)) + " records.")
```

Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

3.8 EXECUÇÃO E COLETA DE DADOS

Assim que todos os algoritmos estavam devidamente implementados, para cada linguagem e grupo de amostra, foi executado 3 vezes o código para, assim, trabalhar-se com a média dos valores apresentados nas execuções. Eram apresentados o primeiro e o último registro para controle posterior no banco, e o tempo de execução, conforme exemplo da linguagem Go no quadro 10, o que diminuiu imprecisões possíveis nos resultados por qualquer variação que possa ser gerada por alguma eventual instabilidade ou inconsistência do sistema.

Pode-se notar também, no quadro, que os algoritmos também exibiam no resultado da execução a quantidade total de elementos que foram salvos na base e o primeiro e último elementos, ordenados pelo campo id, que é um campo sequencial, incrementado de acordo com a inserção de registros.

Quadro 10: Parte da saída da execução do algoritmo em GO

```
*--------- New execution: sample size 1000 ----------*
First record from sample: Kathrine Cox
Last record from sample: Nellie Beale

*--------- 0 execution ---------*
*--------- Synchronous ----------*
Db initialized
[Go] Synchronous implementation tooked 329 milliseconds.
[Go] Processed 1000 records.
Db number of rows: 1000
First record: Kathrine Cox
Last record: Nellie Beale

*--------- Parallel 1 ----------*
Db initialized
[Go] Parallel implementation 1 tooked 78 milliseconds.
[Go] Processed 1000 records.
Db number of rows: 1000
First record: Daniel Shanahan
Last record: Norma Martinez
```

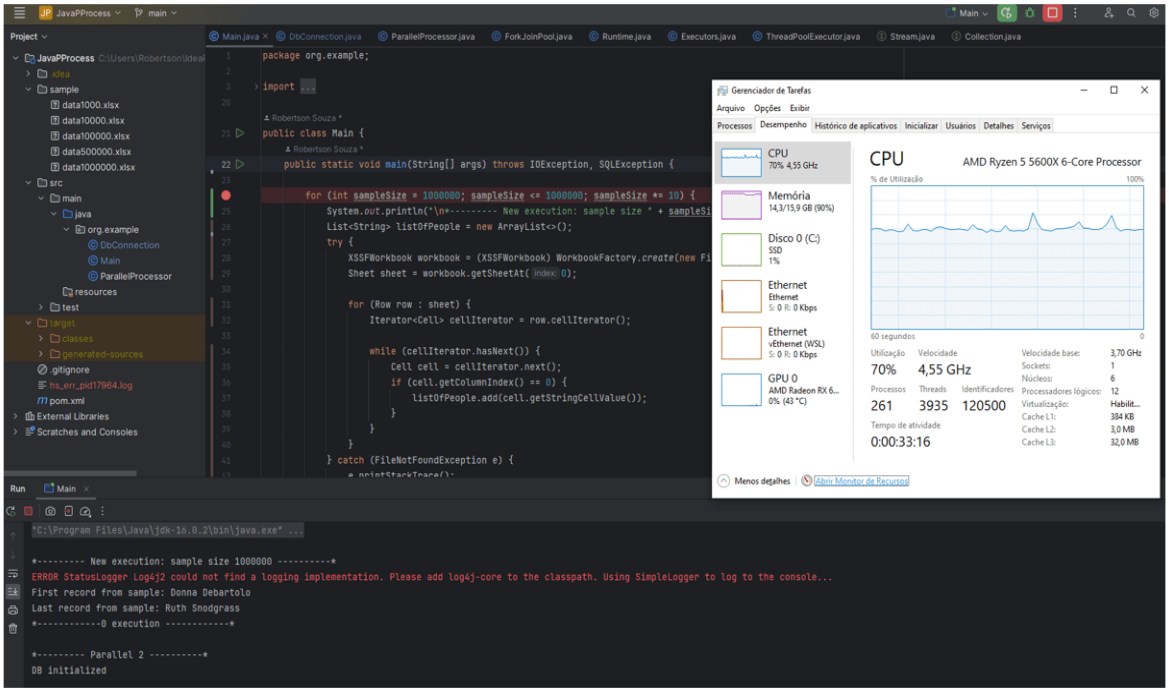
Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

O banco de dados foi a principal ferramenta utilizada para validar se a execução ocorreu com sucesso. Utilizou-se dele para garantir, sempre, que todos os elementos que deveriam ser processados realmente haviam sido, e a forma como isso ocorreu. Conforme comentado, com consultas específicas feitas pelos próprios algoritmos, conseguimos contar os números de registros salvos na base e facilmente entender a ordem em que os mesmos foram salvos, o que permitiu entender se todos os dados foram, então, processados, e se foram lidos sequencialmente, ou não.

No caso da mensuração do uso dos recursos computacionais, o próprio gerenciador do Windows permitiu o acompanhamento através de gráficos, de forma prática, durante a execução dos programas, conforme exemplo na linguagem Java, mostrado na figura 1.

Figura 1: Imagem que mostra forma do acompanhamento do uso de recurso realizado



Fonte: Os autores (2023).

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

4. RESULTADOS

4.1 TABELA DE TEMPO DE EXECUÇÃO

Trabalhou-se com a média aritmética dos valores de tempo apresentados durante as execuções dos algoritmos, conforme a fórmula demonstrada abaixo:

$$média = \frac{(valor1 + valor2 + valor3)}{3}$$

A execução dos algoritmos contemplou tanto o cenário síncrono, quanto o cenário do processamento paralelo, que foi o foco da análise. Foi feito assim para que ficasse mais clara a diferença do resultado das implementações entre as abordagens, a dificuldade para conseguirmos implementar a estratégia em cada cenário e, principalmente, o ganho da performance do paralelismo em relação ao processamento síncrono.

O Quadro 1 apresenta o resultado das médias coletadas das execuções. Os valores apresentados estão em milissegundos e o Input representa o número de elementos que foram processados pelos programas. Também apresentam as complexidades calculadas para cada algoritmo, que se mostraram todas com o valor O(n), onde n é justamente o número do Input, ou seja, o tamanho da amostra dos dados. Esse tamanho afeta diretamente o tempo da execução, onde quanto maior a amostra, mais tempo levará para execução da tarefa.

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Tabela 3: Média de resultados da execução dos algoritmos

Linguagem	Input/ Método	1000	10000	100000	1000000	Complexidade do Algoritmo
GO	síncrono	330	3221,33	32559,33	321003	O (n)
	paralelo	76	488,67	4534,67	45214	O (n)
JAVA	síncrono	784	7613,67	76021,67	771824,67	O (n)
	paralelo	135	1220,33	12289,67	125991,67	O (n)
C#	síncrono	384,33	3584,67	36501	398738,33	O (n)
	paralelo	54	502	5039,33	39672,33	O (n)
PYTHON	síncrono	339,67	3474,67	33353,67	329565,33	O (n)
	paralelo	127	1125	10409,33	106436	O (n)

Fonte: Os autores (2023).

4.2 TABELA SOBRE USO DE RECURSOS COMPUTACIONAIS

Para a análise do uso dos recursos computacionais, comparou-se o estado de descanso da máquina e o estado durante a execução. Os valores de porcentagem de uso de RAM e CPU inicialmente eram:

- 1 % CPU
- 84 % Memória

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Tabela 4: Dados coletados do uso dos recursos computacionais

Linguagem	Input/ Método	CPU	RAM
GO	síncrono	27%	85%
	paralelo	100%	86%
JAVA	síncrono	22%	92%
	paralelo	70%	90%
C#	síncrono	28%	90%
	paralelo	100%	89%
PYTHON	síncrono	21%	80%
	paralelo	100%	81%

Fonte: Os autores (2023).

4.3 ANÁLISE DOS VALORES OBTIDOS

Com os dados coletados nas tabelas anteriores, pode-se fazer algumas afirmativas. Observando-se as complexidades dos algoritmos trabalhados, pode-se notar que todas possuem o mesmo valor, sendo assim, todos os algoritmos são afetados diretamente e evoluem de maneira linear a partir do valor de n, considerando-se n a quantidade de dados enviados para serem processados. Dado esse contexto, entende-se que a evolução do tempo gasto também será linear, ou seja, será um comportamento padrão, sem mudanças relevantes que possam fazer com que esse tempo mude e altere a quantidade da amostra, o que permite que se trabalhe de forma tranquila com os padrões gerados pelos dados.

Com isso, pode-se observar que o C# se destacou na performance de processamento em paralelo no contexto aplicado, mas também que, muito próximo de sua performance, o Go se manteve sempre apto a disputar a posição de menor

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

tempo gasto para realizar as tarefas, até no cenário de processamento síncrono, normalmente com tendência a se destacar na disputa. No caso do paralelismo, pode-se notar uma discrepância maior da performance da linguagem Go e C# diante das outras linguagens, ainda mais conforme se aumenta a carga de entrada de dados a serem trabalhados.

Em relação ao uso de recursos computacionais, pode-se observar que C# e GO tiveram uso de recurso similares, porém, o GO ainda demonstrou poupar um pouco mais o uso da memória durante a mesma tarefa que o C#, e levou uma leve vantagem nesse ponto. Já o JAVA se mostrou a pior no uso dos recursos, usando apenas 70% da CPU da máquina e mais memória RAM que as demais.

5. CONCLUSÃO

Este trabalho conseguiu alcançar satisfatoriamente seus objetivos, demonstrando que uma linguagem de programação otimizada para a arquitetura multinúcleos pode extrair uma melhor performance no processamento em paralelo, assim como se destacaram o Go e C# diante as demais linguagens, no caso do Go, foi criado justamente com o propósito de atender de forma mais eficiente o cenário computacional atual, tanto auxiliando os desenvolvedores na escrita, quanto utilizando de forma mais inteligente os recursos disponibilizados pelos computadores.

Ambas as linguagens se mostraram amigáveis, Go não traz muitas abstrações em seus códigos e por esse ponto acaba exigindo alguns conhecimentos computacionais mais profundos do programador para uma melhor produtividade, como por exemplo a necessidade de trabalhar com ponteiros. Mas ela garante a praticidade e simplicidade em sua escrita, além de outras ferramentas que facilitam algumas das principais atividades do desenvolvedor, e o C# liderou a performance

RC: 150786
Disponível em: https://www.nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

e trouxe uma grande camada de abstração para o algoritmo, tornando-o bem simples também.

Portanto, diante dos pontos avaliados, que foram além da questão escrita e incluíram a performance, temos que C# se mostrou maduro em ambos os pontos, trouxe abstrações práticas e desempenhou boa performance, e temos que Go se manteve muito próximo dele, além de apresentar leve vantagem diante do uso de recursos computacionais.

Com isso, foi possível destacar as melhores linguagens no cenário do paralelismo e demonstrar a importância da escolha de acordo com as necessidades do problema, e mostrar como afetam a solução do desenvolvedor, seja ao facilitar a escrita da solução, seja ao melhorar a eficiência do uso dos recursos computacionais disponíveis para a execução do algoritmo. A performance do programa está diretamente associada a esses resultados, e também é certo que muitas vantagens, inclusive econômicas, são obtidas com o uso adequado dos recursos em grandes aplicações, especialmente as implantadas em grandes empresas, conforme claramente se demonstra a partir do conteúdo apresentado neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

AMAZOM. **O que é Python?**. [s.d.]. Disponível em: <https://aws.amazon.com/pt/what-is/python/>. Acesso em: 12 maio 2023.

BARBOSA, Marco Antonio de Castro; TOSCANI, Laira Vieira. **Metodologia para o cálculo da complexidade de algoritmos e o processo de avaliação das equações de complexidade**. XXXIV Simpósio Brasileiro de Pesquisa Operacional 8 A 11 De Novembro De 2002, Rio De Janeiro/Rj.

DAVID, Lion*; et al.* **Investigating Managed Language Runtime Performance:** Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?. July 11–13, 2022. Disponível em: <https://www.usenix.org/system/files/atc22-lion.pdf>. Acesso em: 21 out. 2023.

GOTARDO, Reginaldo Aparecido. **Linguagens De Programação 1**. ed. Rio de Janeiro: Seses, 2015.

JAVA. **O que é tecnologia Java e por que preciso dela?**. [s.d.]. Disponível em: <https://www.java.com/pt-BR/download/help/whatis_java.html>. Acesso em: 15 out. 2023.

MA, Josué Tza Hsin. **Multicore**. [s.d.]. Disponível em: <https://ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/049180-multicores.pdf>. Acesso em: 9 maio 2023.

MICROSOFT. **O que é o .NET?**. [s.d.]a. Disponível em: <https://dotnet.microsoft.com/pt-br/learn/dotnet/what-is-dotnet>. Acesso em: 13 maio 2023.

MICROSOFT. **Um tour pela linguagem C#.** [s.d.]b. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/>. Acesso em: 10 out. 2023.

NAVAUX, Philippe O. A.. **Introdução ao Processamento Paralelo.** 29 set. 1988. Disponível em: <https://sol.sbc.org.br/index.php/sbac-pad/article/view/23511>. Acesso em: 10 maio 2023.

ORACLE. **JAVA**. Oracle Brasil, [s.d.]. Disponível em: <https://www.oracle.com/br/java/>. Acesso em: 15 out. 2023.

ÖSTLUND, Mikael. **Benchmarking Parallelism and Concurrency in the Encore Programming Language.** out. 2016. Disponível em: <https://www.diva-portal.org/smash/get/diva2:1043738/FULLTEXT02.pdf>. Acesso em: 20 out. 2023.

PIKE, Rob. ***Using Go at Google***. 27 ago. 2007. Disponível em: <https://go.dev/solutions/google/>. Acesso em: 14 mai. 2023.

PYTHON. **The Python Tutoria.** [s.d.]. Disponível em: <https://docs.python.org/3/tutorial/index.html>. Acesso em: 12 maio 2023.

ROSA, Daniel. **O que é a notação Big O:** complexidade de tempo e de espaço. freeCodeCamp, 2021.

SAKURAY, Mônica. **Estudo da Influência dos Parâmetros de Algoritmos Paralelos da Computação Evolutiva no seu Desempenho em Plataformas Multicore.** 2014. Disponível em: <https://repositorio.ufu.br/bitstream/123456789/14340/1/EstudoInfluenciaParametros.pdf>. Acesso em: 10 mai. 2023.

SCAPIN, Victor Hugo Santos. **avaliação do desempenho de bibliotecas para paralelização em arquiteturas multicore:** PTHREAD e OPENMP. 2013. Disponível em: <http://repositorio.utfpr.edu.br/jspui/bitstream/1/7436/1/CP_COADS_2013_1_12.pdf>. Acesso em: 10 mai. 2023.

SUNDAR, B Shyam. **C# vs Rust vs Go. A performance benchmarking in Kubernetes.** feb. 2019. Disponível em: <https://medium.com/@shyamsundarb/c-vs-rust-vs-go-a-performance-benchmarking-in-kubernetes-c303b67b84b5>. Acesso em: 20 out. 2023.

APÊNDICE - NOTA DE RODAPÉ

4. “A CPU (Central Processing Unit) é um chip responsável pelo processamento de dados em um computador, celular e outros dispositivos eletrônicos, funcionando como o “cérebro” desses equipamentos”. Disponível em: <https://tecnoblog.net/responde/o-que-e-cpu-unidade-central-de-processamento>. Acesso em 22 out. 2023.

5. “A memória RAM — Memória de Acesso Aleatório ou *Random Access Memory*, em inglês — permite a leitura e a escrita de arquivos. Ou seja, a sua função é possibilitar que o processador tenha acesso imediato aos dados que deseja, contribuindo para uma maior rapidez e capacidade de resposta das solicitações”. Disponível em: <https://tecnoblog.net/responde/o-que-e-memoria-ram>. Acesso em 22 out. 2023.

Enviado: 12 de junho de 2023.

Aprovado: 12 de dezembro de 2023.

[1] Graduando em Engenharia de Computação. ORCID: https://orcid.org/0009-0001-4799-1687.
[2] Doutora. ORCID: https://orcid.org/0000-0002-9341-0417. Currículo Lattes: http://lattes.cnpq.br/7128829324567785.
[3] Orientador. Mestrado em Ciências da Computação e Matemática Computacional. ORCID: https://orcid.org/0000-0002-0885-0304. Currículo Lattes: http://lattes.cnpq.br/5012184241957733.