Estudo de Técnicas de Programação Concorrente Aplicadas ao Algoritmo de TF-idF

Gabriel Araújo de Souza[1]

[1]Instituto Metrópole Digital (IMD)
Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 1524 – 59.078-970 – Natal – RN – Brasil

gabriel_feg@hotmail.com

Resumo. *Na literatura é possível encontrar diversos algoritmos para mineração e/ou processamento de texto. Entre esses algoritmos, damos destaque ao Term Frequency inverse document Frequency (TF-idF) que informa o quão relevante uma palavra é para um documento no contexto de um conjunto de documentos. No entanto, algoritmos como estes exigem um enorme poder de processamento. Para amenizar este problema, o uso de técnicas de programação concorrente é uma ferramenta de alto poder para ganho de velocidade. Diante disso, este trabalho estuda o desempenho do algoritmo TF-idF ao ser submetido a diferentes implementações com técnicas concorrentes. Testes com as ferramentas Java JCStress, JMH e JMeter foram realizados para constatar o ganho de eficiência dos algoritmos. Uma análise sobre as melhores abordagens concorrentes foi feita com base nos resultados obtidos.*

1. Introdução

Diante dos avanços tecnológicos e do aumento na capacidade de processamento das máquinas, alguns algoritmos de alta complexidade puderam, finalmente, fornecer resultados satisfatórios. Com o lançamento dos processadores *multi-core*, estes algoritmos puderam aumentar seu desempenho através da programação concorrente. Entende-se por isto, dividir as tarefas sequenciais de um algoritmo de forma a serem processadas em paralelo, reduzindo, assim, o tempo de processamento e provendo um ganho significativo de desempenho.

Diversas áreas foram beneficiadas com esses avanços, tais como a da Biotecnologia, Ciência de dados, Aprendizado de máquina e *BigData*. Entre essas grandes áreas, o processamento de linguagem natural (ou simplesmente o processamento de texto) tem ganhado força no mercado. Aplicações como o *Google Assistant*, demonstraram forte avanço nos últimos anos. Dentre os algoritmos de processamento de texto, o *Term Frequency Inverse Document Frequency* (TF-idF), ou em português Frequência de Termo - Frequência Inversa de Documento, merece seu destaque.

O *Term Frequency* (TF) mensura o quão frequente um determinado termo é em relação a um documento em particular. O *Inverse Document Frequency* (idF) mostra o quão frequente um termo é em relação a um conjunto de documentos analisados. Para tanto, se um dado termo ocorre com frequência em um documento particular, mas tem poucas ocorrências nos demais documentos do conjunto explorado, seu valor de TF-idF será alto [Chen 2017]. Sendo assim, este algoritmo calcula o quão relevante um termo é

para um dado documento. Informações como estas podem ser úteis para identificar sobre qual assunto um texto trata.

Diante desse contexto, este trabalho visa implementar diferentes técnicas de programação concorrente para o algoritmo exposto. Uma análise sobre a eficiência do algoritmo em cada técnica será feita. Após a coleta de dados é exposto o ganho de eficiência do algoritmo para cada técnica. Por fim, uma comparação entre as diversas técnicas implementadas é mostrada analisando os pontos positivos e negativos de cada abordagem.

2. Metodologia

O valor de TF-idF de um termo mensura o quão este é relevante para o documento em relação a um conjunto de documentos analisados. Para tanto, esse cálculo pode ser dividido em três etapas: Calcular o valor da frequência de termos, calcular a frequência inversa de documento e realizar o produto entre os dois primeiros.

Define-se o cálculo da frequência de termo da seguinte forma: Seja t um termo qualquer de um documento d. Assuma $N(t)$ como o número de vezes que t ocorre em d e $S(d)$ o número total de termos em d. A frequência de termo para t no documento d, escrita como $\text{TF}(t)_d$, é mostrada abaixo.

$$\text{TF}(t)_d = \frac{N(t)}{S(d)}$$

Seja D um conjunto de documentos de tamanho finito $S(D)$. Seja t um termo qualquer e $F(t)$ o número de documentos em D que há ocorrências de t. Define-se a frequência inversa de Documento (idF) como:

$$\text{idF}(t) = \log\left(\frac{S(D)}{F(t)]}\right)$$

Assim sendo, pode-se pensar no TF-idF como uma matriz $N \times M$ onde N é o número total de termos de todos os documentos e M o número de documentos do conjunto. Um elemento (i, j) diz o valor de TF-idF do termo i para o documento j tal como descrito abaixo:

$$\text{TF-idF}_{i,j} = \text{TF}(i)_j \times \text{idF}(i)$$

Note que uma matriz como essa possuirá diversas células com valores zerados. Isto ocorre pois, existem diversos documentos que não possuem termos em comum, o que zera o valor de TF e consequentemente zera o valor do TF-idF. Com o intuito de reduzir a dimensionalidade dessa matriz, optou-se por comprimi-la armazenando tuplas de três valores. Essa é composta respectivamente pelo termo, nome do documento e valor de TF-idF. Com isso, há uma redução significativa no espaço ocupado pela matriz aumentando a eficiência do algoritmo final.

Com base nessas especificações, implementou-se um software em linguagem Java que constrói a tabela TF-idF, na forma comprimida, para um dado conjunto de documentos. Para os testes, usou-se parte de um *dataset* composto de pequenos textos com críticas sobre filmes [Maas et al. 2011]. Este é composto por 25000 críticas a filmes separados em críticas positiva e negativas, mais 25000 críticas em um conjunto para testes. Destes foram usados os 301 primeiros documentos classificados como críticas positivas.

2.1. Algoritmos

Nesta seção, será descrito os algoritmos implementados, bem como a análise de possíveis implantações de técnicas concorrentes.

Algorithm 1 Descrição do algoritmo principal

```
1: function TFIDF-RUN
2:     READDOCUMENTS( )
3:     CONSTRUCTTERMS( )
4:     TERMFREQUENCY( )
5:     INVERSEDOCUMENT( )
6:     TFIDFTABLE( )
7:
8:     PRINTTABLES( )
```

O Algoritmo 1 mostra o a estrutura principal do algoritmo com as chamadas de funções. Cada função realiza um passo do algoritmo e serão explicadas posteriormente. As operações devem ser realizadas sequencialmente na ordem em que aparecem no código.

O programa utiliza um arquivo auxiliar contendo os links para acessar os documentos a serem lidos. Antes de iniciar, é necessário popular esse arquivo com as referências para os documentos. Um *script* em linguagem Python foi feito para realizar esta operação.

Algorithm 2 Descrição do procedimento READDOCUMENTS

```
1: function READDOCUMENTS(urlDocuments)
2:     reader = FileReader(urlDocuments)
3:     while (line = reader.readLine()) != null do
4:         documents.add(new Document(reader))
```

O Algoritmo 2 mostra a operação de ler os documentos descritos no arquivo auxiliar citado acima. Basicamente ele percorre linha por linha do arquivo, coleta uma URL para um documento e cria um objeto Document com esse parâmetro.

No construtor da classe Document, o documento que a URL corresponde é aberto e todos os termos encontrados são armazenado em um ConcurrentHashMap. Esta classe exige dois parâmetros, uma chave e um valor. A chave foi definida como uma String que representa o termo e para o valor, um inteiro representando o número de ocorrências desse termo encontradas.

Nesta operação há um enorme potencial para aplicação de técnicas concorrentes. É necessário fazer a leitura de diversos arquivos diferentes, isto caracteriza operações de *IO-Intensive*. Para esse tipo de operação, um grande número de *threads* podem ser usados. Para tanto, nas técnicas concorrentes implementadas, fez-se uma análise empírica sobre o quanto o aumento de *threads* usadas influenciaria no desempenho da aplicação. Para este teste, o JMH foi usando com a configuração de 1 *warmup* de cinco iterações, mais

uma execução de 20 iterações após essa fase. Das 20 iterações, anotou-se a média de tempo gasto na operação. Disso, notou-se que 7 *threads* seria um número ideal para esta operação. Os resultados podem ser observados na Figura 1.

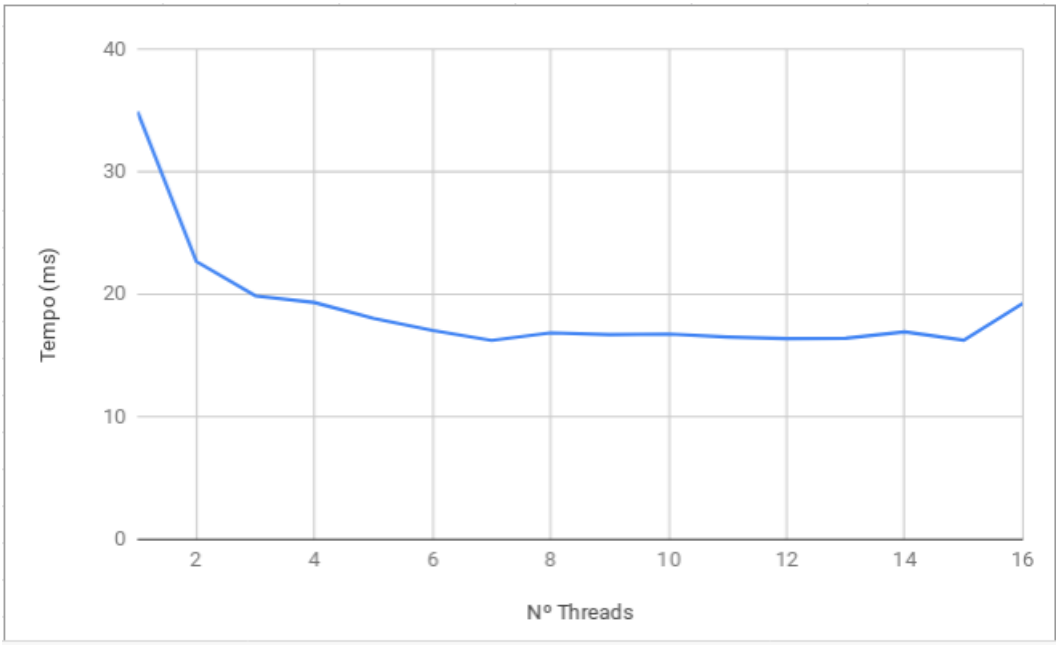

Figura 1. Aumento de desempenho para *IO-Intensive* em relação ao número de *Threads* usadas.

O próximo passo do algoritmo é construir a tabela de termos, para tanto, verifica-se cada documento e adiciona os termos encontrados na tabela de termos. No entanto, muitos desses são considerados como *stop words*, ou seja, palavras que não contribuem para o sentido de um texto, por exemplo, preposições, artigos, conectivos, etc. Assim, é necessário verificar, um a um, se o termo representa uma *stop word*, caso seja, esse termo deve ser desconsiderado. O algoritmo 3 mostra como esse procedimento é realizado.

Algorithm 3 Descrição do procedimento CONSTRUCTTERMS

```
1: function CONSTRUCTTERMS( )
2:     stopWords = StopWordHolder.getInstance()
3:     for Document doc in documents do
4:         for String s in doc.getTerms() do
5:             if not stopWords.isStopWord(s) then
6:                 termsTable.add(s)
```

Um *singleton* foi usado para representar a tabela de palavras definidas como *stop word* (Ver linha 2). O procedimento é bem simples, para cada documento, percorra todos os termos desse documento, se um termo não for uma *stop word* ele é adicionado a tabela de termos (linhas 5 e 6).

Pensando concorrentemente, esse algoritmo pode ser paralelizável por documento. Cada documento é processado independentemente (laço mais externo), portanto,

pode-se usar várias *threads* para processar os documentos de forma paralela.

Algorithm 4 Descrição do procedimento TERMFREQUENCY

```
1: function TERMFREQUENCY( )
2:     for Document doc in documents do
3:         for String s in termsTable do
4:             key = s + " " + doc.name()
5:             numberOfTimesAppears = doc.numberOfOccurrencesTerm(s)
6:             if numberOfTimesAppears != 0 then
7:                 value = numberOfTimesAppears/doc.numberTerms()
8:                 termFrequency.add(key, value)
```

Possuindo a tabela de termos é possível calcular a tabela de frequência de termo. Para armazenar essa tabela de forma comprimida, adotou-se um *ConcurrentHashMap* do Java. A chave de cada valor é dada por uma *String* composta pela concatenação do termo, o símbolo "+"e o nome do documento. O Algoritmo 4 descreve o procedimento.

Para cada documento e para cada termo é verificado se este termo está presente no documento (linha 6). Se essa verificação for positiva, então é calculado o valor de frequência de termo e adicionado na tabela de frequência de termo com a chave criada (linhas 4 e 8). Para tanto, é possível paralelizar o laço mais externo, fazendo linhas cálculo por documento.

Algorithm 5 Descrição do procedimento INVERSEDOCUMENT

```
1: function INVERSEDOCUMENT( )
2:     numberDocs = documents.size()
3:     for String s in termsTable do
4:         count = 0
5:         for Document doc in documents do
6:             if doc.numberOfOccurrences(s) != 0 then count++;
7:         value = log(numDocs/count)
8:         inverseDocument.add(s, value)
```

O Algoritmo 5 mostra o procedimento para cálculo da tabela de frequência inversa de documento. Para cada termo da tabela, percorre-se todos os documentos e verifica-se se aquele documento possui o termo analisado. Um contador armazena a quantidade de documentos com aquele termo encontradas. Por fim, é feito o cálculo de idF e o resultado armazenado na tabela. Esta também utiliza o ConcurrentHashMap do java. Portanto, como a análise de cada termo é independente da outra, então, pode-se paralelizar este algoritmo por termo. Cada termo pode ser avaliado de forma paralela por diversas *threads*.

Algorithm 6 Descrição do procedimento TFIDFTABLE

```
1: function IFIDFTABLE( )
2:     for String s in termsTable do
3:         for Document d in documents do
4:             key = s + " " + doc.name()
5:             if (tfValue = termFrequency.get(key)) != null then
6:                 value = tfValue * inverseDocument.get(key)
7:                 tfIdf.add(key, value)
```

O algoritmo 6 descreve o procedimento final corresponde ao cálculo de TF-idF. Assim como a tabela de frequência de termos, utiliza-se a ideia da chave composta por termo e nome de documento. Percorre-se cada termo e para cada documento verifica se a chave gerada possui valor de TF, se não possuir, essa célula é descartada, caso contrário é realizado o cálculo de TF-idF e armazenado na tabela. Seguindo a lógica dos demais algoritmos, este também pode ser paralelizado, neste caso, por termo.

Sabendo os pontos dos algoritmos que podem ser paralelizados é necessário verificar o ganho de desempenho possível ao implementar técnicas concorrentes. Para essa verificação utiliza-se a Lei de Amdahl definida pela fórmula:

$$Speedup(p) = \frac{1}{s + (1 - s)/p}$$

Onde p é o número de processadores que estão executando o programa e s a porção de código serial. Obviamente, o ganho de desempenho dos algoritmos paralelo implementos dependerá da máquina em que está executando. Os testes foram realizados em um computador com sistema operacional Ubuntu 64-bit, processador Intel Core i5-4200U CPU @ 1,60GHz x 4, memória dedicada de vídeo NVIDIA GeForce GT 740M com 2GB, memória RAM total de 8GB composta por 4 módulos de 2GB em Dual Channel.

O número de *threads* usados nas operações computacionalmente intensivas foi igual ao número de cores da máquina. Este valor pode ser obtida através do comando *Runtime.getRuntime().availableProcessors()* do Java. Usando essa função, o código fica dinâmico e pode ser executado em outros modelos de máquina sem prejuízo no desempenho.

Para auxiliar nos testes realizados, foi utilizado três ferramentas Java JCStress, JMH e JMeter. O primeiro realiza diversos testes, com configurações diferente para assegurar que não haverá condições de corrida nos objetos compartilhados entre as diversas *threads*. O segundo verifica o desempenho a nível *Microbenchmark*, testando o desempenho e cada método e ajudando a encontrar possíveis pontos de paralelismo no código. O último realiza um teste de desempenho da aplicação, podendo simular acessos simultâneos e a resposta do sistema a esses.

3. Técnicas Concorrentes

Nesta seção serão apresentadas as técnicas concorrentes implementadas e os testes de concorrência realizadas através da ferramenta JCStress.

Para os cálculos de ganho de desempenho, realizou-se a seguinte metodologia. Tem-se 6 funções que possuem aproximadamente a mesma quantidade de código, sendo assim, cada função representa uma parcela aproximada de 16% do código (Considerando outras pequenas parcelas de código que não estão presente nas funções descritas). Com base nessa estimativa, fica fácil de calcular o ganho teórico de desempenho de cada função paralelizada.

3.1. Mutex

A técnica mais clássica que pode ser pensada para utilizar concorrência é o Mutex. Esse consiste em proteger um objeto compartilhado entre diversas *threads* por meio de um *lock* adequado. Para tanto, nos algoritmos analisados encontramos duas abordagens diferentes de paralelismo, por documentos ou por termos. Quando há essa divisão, é importante garantir que duas *threads* diferentes nunca possam processar o mesmo documento ou o mesmo tempo.

A abordagem pensada para a implementação do Mutex foi compartilhar um *ArrayList* com os objetos (termos ou documentos) que devem ser processados de forma paralela. O que deseja-se é avançar posição a posição do *array* de forma paralela e garantir que duas *threads* diferentes nunca irão acessar a mesma posição desse *array*. Para isso, criou-se um contador protegido por um Mutex. Executou-se um teste no JCStress para verificar se três *threads* diferentes, ao acessar esse contador, pegariam posições diferentes. Com base nos resultados, pôde-se notar que o Mutex foi implementado de forma correta e não houve problemas de concorrência. Os resultados podem ser vistos na Figura 2. As configurações usadas nos testes do JCStress podem ser vistas abaixo.

- TC 1 JVM options: [-Dfile.encoding=UTF-8] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
- TC 2 JVM options: [-Dfile.encoding=UTF-8, -XX:-TieredCompilation] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
- TC 3 JVM options: [-Dfile.encoding=UTF-8, -XX:TieredStopAtLevel=1] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
- TC 4 JVM options: [-Dfile.encoding=UTF-8, -Xint] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)

Observed state	TC 1	TC 2	TC 3	TC 4	
0, 1, 2	3227965	3381212	5183697	908879	ACCEPTABLE
0, 2, 1	2818830	2911564	1358620	181157	ACCEPTABLE
1, 0, 2	2292631	2053280	4685203	426370	ACCEPTABLE
1, 2, 0	2325078	2261518	3474415	408858	ACCEPTABLE
2, 0, 1	2043433	1781865	735525	105884	ACCEPTABLE
2, 1, 0	2423684	2187542	2755011	449313	ACCEPTABLE
	OK	OK	OK	OK	

Figura 2. Resultados do JCStress para o contador com Mutex

Das funções disponíveis, apenas a de escrever as tabelas geradas não pôde ser paralelizada (printTables). Para as demais funções, pôde-se paralelizar o laço mais externo

de cada uma das funções, ou seja, praticamente 50% de cada método, logo, isto resulta num total de 40% de código paralelizado. Utilizando a Lei de Amdahl, sabendo que o computador utilizado possui 4 núcleos de processamento, tem-se um ganho de desempenho esperado é de de

$$Speedup(4) = \frac{1}{0,6 + (0,4)/4} \approx 1,42$$

3.2. Semáforo

De forma similar ao Mutex, podemos utilizar um Semáforo para proteger os objetos compartilhados. A abordagem pensada foi um pouco diferente da usada no Mutex. Ao invés de usar um contador que informa a posição do *Array* que deve ser acessado, criou-se uma pilha com os objetos a serem processados (documentos ou termos), assim, enquanto a pilha não estiver vazia é solicitado o elemento no topo para ser processado.

Nesse exemplo da pilha foi criado um semáforo para proteger a região crítica de acesso a pilha, ou seja, a operação recuperar o elemento do topo da pilha e removê-lo. Para tanto, uma *thread* solicita o próximo elemento que ele deve processar, ao chamar o método do semáforo, ele ativa uma requisição impedindo que outras *threads* entre na região crítica, acessa e remove o elemento do topo da pilha, libera o *lock* e retorna o objeto acessado. Se a pilha estiver vazia, um objeto nulo é retornado e a *thread* para de executar. Duas versões de semáforo foram criados, um para proteger uma pilha de Documentos e outro para proteger uma pilha de termos. Os resultados do JCStress podem ser vistos nas Figuras 3 e 4, em ambos os casos foi verificado se não haveria o problema de duas *threads* diferentes receberem o mesmo objeto. Os objetos foram classificados com "a"ou "b". As configurações usadas pelo JCStress são as mesmas mostradas anteriormente.

Observed state	TC 1	TC 2	TC 3	TC 4	
a, b	5559680	4024901	4052920	128580	ACCEPTABLE
b, a	4363511	5376240	3157121	830101	ACCEPTABLE
	OK	OK	OK	OK	

Figura 3. Resultados do JCStress para o Semáforo de documentos

Observed state	TC 1	TC 2	TC 3	TC 4	
a, b	0	7075330	1359970	316990	ACCEPTABLE
b, a	9599581	2058501	5795961	563951	ACCEPTABLE
	OK	OK	OK	OK	

Figura 4. Resultados do JCStress para o Semáforo de Termos

A proporção de código paralelizado foi a mesma que a do Mutex, o que nos dá um ganho esperado de desempenho de $1,42$ ou 42% .

4. Resultados

Implementadas as técnicas, além dos testes com JCStress, ainda foi realizado teste de *Microbenchmark* e *Macrobenchmark*. No primeiro utilizou-se o JMH e verificou o desempenho de cada função principal para as diversas técnicas implementadas. No segundo, verificou-se o desempenho do sistema com cada técnica com os seguintes parâmetros no JMeter: Simulando 4 requisições simultâneas e 100 iterações.

A Tabela 1 é constituída da seguinte forma: Nas linhas temos as funções individuais do algoritmo e as colunas representam os resultados obtidos para cada algoritmo. Cada coluna principal possui os atributos Avg e $\%$ que significam respectivamente a média de tempo de execução para determinada função e a porcentagem do ganho de desempenho em relação ao algoritmo sequencial. Note que por causa desse fato, no algoritmo sequencial, todos os valores desta coluna são 0%, portanto foram omitidos.

	Sequencial	Mutex		Semáforo	
Função	Avg	Avg	%	Avg	%
readDocuments	30,290	16,701	44,86%	**16,309**	46,15%
constructTerms	**7,849**	11,883	-51,39%	11,446	-45,82%
termFrequency	279,359	**125,712**	55,00%	136,358	51,18%
inverseDocument	139,172	**65,487**	52,94%	82,824	40,48%
tfidfTable	327,677	340,554	-3,93%	**183,268**	44,07 %
printTables	110,786	112,078	-1,16%	**102,579**	7,40%
Total	895,133	672,415	24,88%	**532,784**	40,47%

Tabela 1. Resultado de testes Microbenchmark com JMH

Na tabela 1 vemos os resultados dos testes com o JMH. Em destaques estão os melhores resultados de cada método. Notou-se que o método *constructTerms* sequencial, obteve êxito em relação as técnicas concorrentes. Um dos possíveis motivos para isso seja o *singleton* utilizado neste método para verificar se um termo é uma *stop word*. Isso decorre pois o método chamado possui um mutex para evitar erros de concorrência, assim, nas técnicas concorrentes, essa função estaria usando dois Mutex.

Nota-se que o Mutex, para o cálculo da tabela TF-idF não possui ganho significativo em relação ao sequencial. No entanto, se considerarmos o algoritmo como um todo houve um ganho de desempenho significativo $(24,88\%)$. No geral, é possível perceber que a técnica de Semáforo obteve os melhores resultados, tanto por método, como no algoritmo como um todo, proporcionando um ganho total de $40,47\%$, bem próximo do previsto pela Lei de Amdahl.

Com base nos resultados obtidos neste teste, decidiu-se construir um algoritmo híbrido em que cada implementação de método usará a técnica que obteve melhor resultado. Assim, esse novo algoritmo possui uma mistura do algoritmo sequencial, Mutex e Semáforo. Como resultado final, conseguimos um ganho de desempenho total de $43,36\%$ em relação ao algoritmo sequencial. Os detalhes podem ser consultados na Tabela 2 que segue os mesmos critérios de representatividade da Tabela 1.

Função	Avg	%
readDocuments	16,463	45,65%
constructTerms	8,490	-8,16%
termFrequency	128,765	53,90%
inverseDocument	64,511	53,64%
tfidfTable	167,111	49,00%
printTables	121,633	-9,79%
Total	506,979	43,36%

Tabela 2. Resultado de testes Microbenchmark com JMH para a técnica Híbrida

O teste de desempenho através do JMeter pode ser observado na Tabela 3. Cada linha representa uma técnica concorrente diferente, foram incluídos os testes para o algoritmo híbrido gerado. A coluna "Amostras"representa o total de testes realizados, ele é dado pelo número de solicitações simultâneas vezes o número de iterações. As colunas Média, Min, Max representam respectivamente o tempo médio, mínimo e máximo de resposta para atender as requisições. A coluna DP representa o desvio padrão das amostras de tempo coletadas e a última coluna a quantidade de requisições atendidas por segundo. Os parâmetros usados para os testes foram duas *threads* simultâneas iniciação a cada 1 segundo e 200 iterações.

Algoritmo	Amostras	Média	Min	Max	DP	Vazão
Sequencial	400	1172	989,25	2539	113,60	1,70
Mutex	400	1050	692	2834	152,66	1,89
Semáforo	400	1076	582	3326	191,53	1,85
Híbrido	400	1012	513	2793	177,03	1,96

Tabela 3. Testes de desempenho com JMeter

5. Conclusão

Diante dos testes realizados, pôde-se constar a real força que a programação concorrente possui. Um algoritmo como o TF-idF trata conjunto de dados extremamente grandes (bem maiores que o utilizado neste trabalho), portanto, qualquer forma de aumento de desempenho conta para algoritmos dessa proporção.

Os testes mostraram que técnicas bem simples como Mutex e Semáforo podem proporcionar ganhos significativos de desempenho quando aplicados corretamente. O uso de ferramentas apropriadas puderam comprovar os fatos aqui afirmados. Sobre as ferramentas de testes, estas desempenham enorme importância para auxiliar no desenvolvimento de códigos concorrentes, e poder apresentar os resultados obtidos por elas representam uma enorme contribuição para a literatura.

Constatou-se uma pequena divergência entre os testes realizados pela ferramenta JVM e JMeter entre as técnicas Mutex e Semáforo, dessa forma, é possível detectar um ponto de revisão dos códigos fontes e/ou testes realizados. No entanto, de uma maneira geral, os testes dessas ferramentas condizem com a realidade observada empiricamente. Por fim, este trabalho demonstrou a importância de se estudar técnicas de programação

concorrente e aperfeiçoar os softwares desenvolvidos. Este trabalho serve como referência e motivação para que futuras pesquisas possam ser desenvolvidas na área.

Referências

Tf-idf :: a single-page tutorial - information retrieval and text mining. `http://www.tfidf.com/`. Acessado em: 03-03-2019.

Chen, C.-H. (2017). Improved tfidf in big news retrieval: An empirical study. *Pattern Recognition Letters*, 93:113–122.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.